$\langle\,Version\,\rangle\equiv$
```
2.0.0
```
$\langle\,Date\,\rangle\equiv$
```
Mon Apr 12 2010
```

WHIZARD[1]

Wolfgang Kilian,[2] Thorsten Ohl,[3] Jürgen Reuter[4]

Version 2.0.0, Apr 12 2010

---

[1]The original meaning of the acronym is *W, Higgs, Z, And Respective Decays*. The current program is much more than that, however.

[2]e-mail: `kilian@hep.physik.uni-siegen.de`

[3]e-mail: `ohl@physik.uni-wuerzburg.de`

[4]e-mail: `juergen.reuter@physik.uni-freiburg.de`

Abstract

WHIZARD is an application of the VAMP algorithm: Adaptive multi-channel integration and event generation. The bare VAMP library is augmented by modules for Lorentz algebra, particles, phase space, etc., such that physical processes with arbitrary complex final states [well, in principle. . .] can be integrated and *unweighted* events be generated.

Contents

Chapter 1

Changes

2.0.0 – Most of the WHIZARD 1 code has either been revised or rewritten

    – Completely new user interface (script language)

    – New implementation of density matrix evaluation (which is the very core)

Chapter 2

Tools

This part contains modules needed for auxiliary purposes:

limits Compile-time (integer) constants: fixed array sizes, input field lengths, and such. Any module that uses such constants has to access them via **limits**.

mpi90 Parallel execution (currently disabled!). This module contains dummy replacements for the message passing interface subroutines.

clock Store and handle dates and times

diagnostics Error and diagnostic message handling. Any messages and errors issued by WHIZARD functions are handled by the subroutines in this module, if possible.

files Files and auxiliary routines that do not belong anywhere else.

permutations Handle permutations of integers.

unix_args Parse the command-line arguments and options. This part is system-dependent, and we provide a replacement if the functions are not available.

2.1 Preliminaries

The WHIZARD file header:

⟨*File header*⟩≡
```
  ! WHIZARD ⟨Version⟩ ⟨Date⟩
  !
  ! (C) 1999-2010 by
  !     Wolfgang Kilian <kilian@hep.physik.uni-siegen.de>
  !     Thorsten Ohl <ohl@physik.uni-wuerzburg.de>
  !     Juergen Reuter <juergen.reuter@physik.uni-freiburg.de>
  !     with contributions by Christian Speckner, Sebastian Schmidt,
  !     Daniel Wiesler, Felix Braam
  !
  ! WHIZARD is free software; you can redistribute it and/or modify it
  ! under the terms of the GNU General Public License as published by
  ! the Free Software Foundation; either version 2, or (at your option)
```

```
! any later version.
!
! WHIZARD is distributed in the hope that it will be useful, but
! WITHOUT ANY WARRANTY; without even the implied warranty of
! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
! GNU General Public License for more details.
!
! You should have received a copy of the GNU General Public License
! along with this program; if not, write to the Free Software
! Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This file has been stripped of most comments.  For documentation, refer
! to the source 'whizard.nw'
```

We are strict with our names:

⟨*Standard module head*⟩≡
```
    implicit none
    private
```

This is the way to envoke the kinds module (not contained in this source)

⟨*Use kinds*⟩≡
```
    use kinds, only: default !NODEP!
```

And we make heavy use of variable-length strings

⟨*Use strings*⟩≡
```
    use iso_varying_string, string_t => varying_string !NODEP!
```

Some parameters (buffer sizes etc.) are hardcoded. They are collected in this module:

⟨`limits.f90`⟩≡
  ⟨*File header*⟩

```
  module limits
```

  ⟨*Standard module head*⟩

  ⟨*Limits: public parameters*⟩

```
  end module limits
```

The version string is used for checking files. Note that the string length MUST NOT be changed, because reading binary files relies on it.

⟨*Limits: public parameters*⟩≡
```
    integer, parameter, public :: VERSION_STRLEN = 255
    character(len=VERSION_STRLEN), parameter, public :: &
        & VERSION_STRING = "WHIZARD version ⟨Version⟩ (⟨Date⟩)"
```


2.2   Parallelization

This section has been removed, it was never used even in WHIZARD1. Parallelization should be reimplemented from scratch.

2.3 File utilities

This module provides tools for finding a free unit, filename handling: directory prefixes and extensions, default filenames. Furthermore, the `upper_case` and `lower_case` functions and a tool for formatting numbers in TeX style.

⟨`file_utils.f90`⟩≡
  ⟨*File header*⟩

```
  module file_utils

    use iso_fortran_env, only: stdout => output_unit !NODEP!
```
  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
```
    use limits, only: MIN_UNIT, MAX_UNIT !NODEP!
    use limits, only: DEFAULT_FILENAME, FILENAME_LEN !NODEP!
```

  ⟨*Standard module head*⟩

  ⟨*File utils: public*⟩

  ⟨*File utils: interfaces*⟩

```
  contains
```

  ⟨*File utils: procedures*⟩

```
  end module file_utils
```

2.3.1 Finding an I/O unit

Fortran 95 (even Fortran 2003) has no notion of implicit I/O units. Therefore, we have to find a free unit by trial and error.

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: MIN_UNIT = 11, MAX_UNIT = 99
```

⟨*File utils: public*⟩≡
```
    public :: free_unit
```

⟨*File utils: procedures*⟩≡
```
    function free_unit () result (unit)
      integer :: unit
      logical :: exists, is_open
      integer :: i, status
      do i = MIN_UNIT, MAX_UNIT
         inquire (unit=i, exist=exists, opened=is_open, iostat=status)
         if (status == 0) then
            if (exists .and. .not. is_open) then
               unit = i; return
            end if
         end if
      end do
      unit = -1
    end function free_unit
```

14

Return the given unit, if present, otherwise the default STDOUT unit.

⟨*File utils: public*⟩+≡
```
    public :: output_unit
```

⟨*File utils: procedures*⟩+≡
```
    function output_unit (unit) result (u)
      integer, intent(in), optional :: unit
      integer :: u
      if (present (unit)) then
         u = unit
      else
         u = stdout
      end if
    end function output_unit
```

This activates the ubiquitous functions `free_unit` and `output_unit`:

⟨*Use file utils*⟩≡
```
    use file_utils !NODEP!
```

Flush all units possibly activated by `free_unit`

⟨*File utils: public*⟩+≡
```
    public :: flush_all
```

⟨*File utils: procedures*⟩+≡
```
    subroutine flush_all ()
      integer :: u
      do u = MIN_UNIT, MAX_UNIT
         flush (u)
      end do
    end subroutine flush_all
```

2.3.2   Filename handling

Choose a filename. If the first one is empty, take the second one. (Of course, this function works for any string.)

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: FILENAME_LEN = 256
```

⟨*File utils: public*⟩+≡
```
    public :: choose_filename
```

⟨*File utils: procedures*⟩+≡
```
    function choose_filename (file, default_file) result (chosen_file)
      character(len=*), intent(in) :: file, default_file
      character(len=FILENAME_LEN) :: chosen_file
      if (len_trim (file) == 0) then
         chosen_file = default_file
      else
         chosen_file = file
      end if
    end function choose_filename
```

Concat filename with dot(s) and extension(s). The directory prefix is prepended only if the filename is relative

⟨*File utils: public routines*⟩≡
```
    public :: concat
```

⟨*File utils: interfaces*⟩≡
```
    interface concat
       module procedure concat_two, concat_three
    end interface
```

⟨*File utils: procedures*⟩+≡
```
    function concat_two (prefix, filename, extension) result (file)
      character(len=*) :: prefix, filename, extension
      character(len=FILENAME_LEN) :: file
      file = trim(filename)//"."//trim(extension)
      if (prefix /= "" .and. filename(1:1) /= "/" .and. filename(1:2) /= "./") &
           & file = trim(prefix) // "/" // file
    end function concat_two

    function concat_three (prefix, filename, process_id, extension) result (file)
      character(len=*) :: prefix, filename, process_id, extension
      character(len=FILENAME_LEN) :: file
      file = trim(filename)//"."//trim(process_id)//"."//trim(extension)
      if (prefix /= "" .and. filename(1:1) /= "/" .and. filename(1:2) /= "./") &
           & file = trim(prefix) // "/" // file
    end function concat_three
```

Select a file (to open later for reading). If the first one does not exist, try the default filename. Return the filename with extension appended. If the default file does not exist either, return the empty string.

⟨*Limits: public parameters*⟩+≡
```
    character(len=*), parameter, public :: DEFAULT_FILENAME = "whizard"
```

⟨*File utils: public*⟩+≡
```
    public :: file_exists_else_default
```

⟨*File utils: procedures*⟩+≡
```
    function file_exists_else_default (prefix, filename, extension) result (file)
      character(len=*) :: prefix, filename, extension
      character(len=FILENAME_LEN) :: file
      logical :: exist
      file = concat (prefix, filename, extension)
      inquire (file=trim(file), exist=exist)
      if (.not.exist) then
         file = concat (prefix, DEFAULT_FILENAME, extension)
         inquire (file=trim(file), exist=exist)
         if (.not.exist)  then
            file = ""
         end if
      end if
    end function file_exists_else_default
```

2.3.3 String auxiliary functions

These are, unfortunately, not part of Fortran.

⟨*File utils: public*⟩+≡

```
    public :: upper_case
    public :: lower_case
```

⟨*File utils: interfaces*⟩+≡

```
    interface upper_case
       module procedure upper_case_char, upper_case_string
    end interface
    interface lower_case
       module procedure lower_case_char, lower_case_string
    end interface
```

⟨*File utils: procedures*⟩+≡

```
    function upper_case_char (string) result (new_string)
      character(*), intent(in) :: string
      character(len(string)) :: new_string
      integer :: pos, code
      integer, parameter :: offset = ichar('A')-ichar('a')
      do pos = 1, len (string)
         code = ichar (string(pos:pos))
         select case (code)
         case (ichar('a'):ichar('z'))
            new_string(pos:pos) = char (code + offset)
         case default
            new_string(pos:pos) = string(pos:pos)
         end select
      end do
    end function upper_case_char

    function lower_case_char (string) result (new_string)
      character(*), intent(in) :: string
      character(len(string)) :: new_string
      integer :: pos, code
      integer, parameter :: offset = ichar('a')-ichar('A')
      do pos = 1, len (string)
         code = ichar (string(pos:pos))
         select case (code)
         case (ichar('A'):ichar('Z'))
            new_string(pos:pos) = char (code + offset)
         case default
            new_string(pos:pos) = string(pos:pos)
         end select
      end do
    end function lower_case_char

    function upper_case_string (string) result (new_string)
      type(string_t), intent(in) :: string
      type(string_t) :: new_string
      new_string = upper_case_char (char (string))
    end function upper_case_string

    function lower_case_string (string) result (new_string)
```

```
      type(string_t), intent(in) :: string
      type(string_t) :: new_string
      new_string = lower_case_char (char (string))
   end function lower_case_string
```

2.3.4 Formatting numbers

Format a number with n significant digits.

⟨*File utils: public*⟩+≡
```
    public :: tex_format
```

⟨*File utils: procedures*⟩+≡
```
    function tex_format (rval, n_digits) result (string)
      type(string_t) :: string
      real(default), intent(in) :: rval
      integer, intent(in) :: n_digits
      integer :: e, n, w, d
      real(default) :: absval
      real(default) :: mantissa
      character :: sign
      character(20) :: format
      character(80) :: cstr
      n = min (abs (n_digits), 16)
      if (rval == 0) then
         string = "0"
      else
         absval = abs (rval)
         e = log10 (absval)
         if (rval < 0) then
            sign = "-"
         else
            sign = ""
         end if
         select case (e)
         case (:-3)
            d = max (n - 1, 0)
            w = max (d + 2, 2)
            write (format, "('(F',I0,'.',I0,',',A,I0,A)')")  w, d
            mantissa = absval * 10._default ** (1 - e)
            write (cstr, fmt=format)  mantissa, "\times 10^{", e - 1, "}"
         case (-2:0)
            d = max (n - e, 1 - e)
            w = max (d + e + 2, d + 2)
            write (format, "('(F',I0,'.',I0,')')")  w, d
            write (cstr, fmt=format)  absval
         case (1:2)
            d = max (n - e - 1, -e, 0)
            w = max (d + e + 2, d + 2, e + 2)
            write (format, "('(F',I0,'.',I0,')')")  w, d
            write (cstr, fmt=format)  absval
         case default
            d = max (n - 1, 0)
            w = max (d + 2, 2)
```

```
        write (format, "('(F',I0,'.',I0,',A,I0,A)')")  w, d
        mantissa = absval * 10._default ** (- e)
        write (cstr, fmt=format)  mantissa, "\times 10^{", e, "}"
     end select
     string = sign // trim (cstr)
  end if
end function tex_format
```

2.4 Operating-system interface

For specific purposes, we need direct access to the OS (system calls). This is,
of course, system dependent. The current version is valid for GNU/Linux; we
expect to use a preprocessor for this module if different OSs are to be supported.

The current implementation lacks error handling.

⟨os_interface.f90⟩≡
  ⟨File header⟩

```
  module os_interface

    use iso_c_binding !NODEP!
```
  ⟨Use strings⟩
  ⟨Use file utils⟩
```
    use system_dependencies !NODEP!
    use limits, only: DLERROR_LEN, ENVVAR_LEN !NODEP!
    use diagnostics !NODEP!
```

  ⟨Standard module head⟩

  ⟨OS interface: public⟩

  ⟨OS interface: types⟩

  ⟨OS interface: interfaces⟩

```
  contains
```

  ⟨OS interface: procedures⟩

```
  end module os_interface
```

2.4.1 Path variables

This is a transparent container for storing user-defined path variables.

⟨OS interface: public⟩≡
```
    public :: paths_t
```

⟨OS interface: types⟩≡
```
    type :: paths_t
       type(string_t) :: prefix
       type(string_t) :: exec_prefix
       type(string_t) :: bindir
```

```
      type(string_t) :: libdir
      type(string_t) :: includedir
      type(string_t) :: datarootdir
      type(string_t) :: localprefix
   end type paths_t
```

⟨*OS interface: public*⟩+≡
```
    public :: paths_init
```

⟨*OS interface: procedures*⟩≡
```
    subroutine paths_init (paths)
      type(paths_t), intent(out) :: paths
      paths%prefix = ""
      paths%exec_prefix = ""
      paths%bindir = ""
      paths%libdir = ""
      paths%includedir = ""
      paths%datarootdir = ""
      paths%localprefix = ""
    end subroutine paths_init
```

2.4.2 System dependencies

We store all potentially system- and user/run-dependent data in a transparent
container. This includes compiler/linker names and flags, file extensions, etc.

⟨*OS interface: public*⟩+≡
```
    public :: os_data_t
```

⟨*OS interface: types*⟩+≡
```
    type :: os_data_t
       logical :: use_libtool
       logical :: use_testfiles
       type(string_t) :: fc
       type(string_t) :: fcflags
       type(string_t) :: fcflags_pic
       type(string_t) :: fc_src_ext
       type(string_t) :: obj_ext
       type(string_t) :: ld
       type(string_t) :: ldflags
       type(string_t) :: ldflags_so
       type(string_t) :: ldflags_static
       type(string_t) :: shlib_ext
       type(string_t) :: prefix
       type(string_t) :: exec_prefix
       type(string_t) :: bindir
       type(string_t) :: libdir
       type(string_t) :: includedir
       type(string_t) :: datarootdir
       type(string_t) :: whizard_omega_binpath
       type(string_t) :: whizard_includes
       type(string_t) :: whizard_ldflags
       type(string_t) :: whizard_libtool
       type(string_t) :: whizard_modelpath
```

```
    type(string_t) :: whizard_models_libpath
    type(string_t) :: whizard_susypath
    type(string_t) :: whizard_gmlpath
    type(string_t) :: whizard_cutspath
    type(string_t) :: whizard_texpath
    type(string_t) :: whizard_testdatapath
    type(string_t) :: whizard_modelpath_local
    type(string_t) :: whizard_models_libpath_local
    type(string_t) :: whizard_omega_binpath_local
    logical :: event_analysis_ps  = .false.
    logical :: event_analysis_pdf = .false.
    type(string_t) :: latex
    type(string_t) :: gml
    type(string_t) :: dvips
    type(string_t) :: ps2pdf
  end type os_data_t
```

Since all are allocatable strings, explicit initialization is necessary.

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: ENVVAR_LEN = 1000
```

⟨*OS interface: public*⟩+≡
```
    public :: os_data_init
```

⟨*OS interface: procedures*⟩+≡
```
    subroutine os_data_init (os_data, paths)
      type(os_data_t), intent(out) :: os_data
      type(paths_t), intent(in), optional :: paths
      character(len=ENVVAR_LEN) :: home
      type(string_t) :: localprefix, local_includes
      os_data%use_libtool = .true.
      inquire (file = "TESTFLAG", exist = os_data%use_testfiles)
      call get_environment_variable ("HOME", home)
      if (paths%localprefix == "") then
         localprefix = trim (home) // "/.whizard"
      else
         localprefix = paths%localprefix
      end if
      local_includes = localprefix // "/lib/whizard/mod/models"
      os_data%whizard_modelpath_local = localprefix // "/share/whizard/models"
      os_data%whizard_models_libpath_local = localprefix // "/lib/whizard/models"
      os_data%whizard_omega_binpath_local = localprefix // "/bin"
      os_data%fc            = DEFAULT_FC
      os_data%fcflags       = DEFAULT_FCFLAGS
      os_data%fcflags_pic   = DEFAULT_FCFLAGS_PIC
      os_data%fc_src_ext    = DEFAULT_FC_SRC_EXT
      os_data%obj_ext       = DEFAULT_OBJ_EXT
      os_data%ld            = DEFAULT_LD
      os_data%ldflags       = DEFAULT_LDFLAGS
      os_data%ldflags_so    = DEFAULT_LDFLAGS_SO
      os_data%ldflags_static = DEFAULT_LDFLAGS_STATIC
      os_data%shlib_ext     = DEFAULT_SHLIB_EXT
      os_data%prefix      = PREFIX
      os_data%exec_prefix = EXEC_PREFIX
      os_data%bindir      = BINDIR
```

```
   os_data%libdir     = LIBDIR
   os_data%includedir = INCLUDEDIR
   os_data%datarootdir = DATAROOTDIR
   if (present (paths)) then
      if (paths%prefix      /= "")  os_data%prefix      = paths%prefix
      if (paths%exec_prefix /= "")  os_data%exec_prefix = paths%prefix
      if (paths%bindir      /= "")  os_data%bindir      = paths%prefix
      if (paths%libdir      /= "")  os_data%libdir      = paths%prefix
      if (paths%includedir  /= "")  os_data%includedir  = paths%prefix
      if (paths%datarootdir /= "")  os_data%datarootdir = paths%prefix
   end if
   if (os_data%use_testfiles) then
      os_data%whizard_omega_binpath  = WHIZARD_TEST_OMEGA_BINPATH
      os_data%whizard_includes       = WHIZARD_TEST_INCLUDES
      os_data%whizard_ldflags        = WHIZARD_TEST_LDFLAGS
      os_data%whizard_libtool        = WHIZARD_LIBTOOL_TEST
      os_data%whizard_modelpath      = WHIZARD_TEST_MODELPATH
      os_data%whizard_models_libpath = WHIZARD_TEST_MODELS_LIBPATH
      os_data%whizard_susypath       = WHIZARD_TEST_SUSYPATH
      os_data%whizard_gmlpath        = WHIZARD_TEST_GMLPATH
      os_data%whizard_cutspath       = WHIZARD_TEST_CUTSPATH
      os_data%whizard_texpath        = WHIZARD_TEST_TEXPATH
      os_data%whizard_testdatapath   = WHIZARD_TEST_TESTDATAPATH
   else
      if (os_dir_exist (local_includes)) then
         os_data%whizard_includes = "-I" // local_includes // " "// &
            WHIZARD_INCLUDES
      else
         os_data%whizard_includes = WHIZARD_INCLUDES
      end if
      os_data%whizard_omega_binpath  = WHIZARD_OMEGA_BINPATH
      os_data%whizard_ldflags        = WHIZARD_LDFLAGS
      os_data%whizard_libtool        = WHIZARD_LIBTOOL
      os_data%whizard_modelpath      = WHIZARD_MODELPATH
      os_data%whizard_models_libpath = WHIZARD_MODELS_LIBPATH
      os_data%whizard_susypath       = WHIZARD_SUSYPATH
      os_data%whizard_gmlpath        = WHIZARD_GMLPATH
      os_data%whizard_cutspath       = WHIZARD_CUTSPATH
      os_data%whizard_texpath        = WHIZARD_TEXPATH
      os_data%whizard_testdatapath   = WHIZARD_TESTDATAPATH
   end if
   os_data%event_analysis_ps  = EVENT_ANALYSIS_PS  == "yes"
   os_data%event_analysis_pdf = EVENT_ANALYSIS_PDF == "yes"
   os_data%latex  = PRG_LATEX
   os_data%gml    = os_data%whizard_gmlpath // "/gml"
   os_data%dvips  = PRG_DVIPS
   os_data%ps2pdf = PRG_PS2PDF
   call os_data_expand_paths (os_data)
end subroutine os_data_init
```

Replace occurences of GNU path variables (such as `${prefix}`) by their values.
Do this for all strings that could depend on them, and do the replacement in
reverse order, since the path variables may be defined in terms of each other.

⟨*OS interface: procedures*⟩+≡

```
subroutine os_data_expand_paths (os_data)
  type(os_data_t), intent(inout) :: os_data
  integer, parameter :: N_VARIABLES = 6
  type(string_t), dimension(N_VARIABLES) :: variable, value
  variable(1) = "${prefix}";       value(1) = os_data%prefix
  variable(2) = "${exec_prefix}";  value(2) = os_data%exec_prefix
  variable(3) = "${bindir}";       value(3) = os_data%bindir
  variable(4) = "${libdir}";       value(4) = os_data%libdir
  variable(5) = "${includedir}";   value(5) = os_data%includedir
  variable(6) = "${datarootdir}";  value(6) = os_data%datarootdir
  call expand_paths (os_data%whizard_omega_binpath)
  call expand_paths (os_data%whizard_includes)
  call expand_paths (os_data%whizard_ldflags)
  call expand_paths (os_data%whizard_libtool)
  call expand_paths (os_data%whizard_modelpath)
  call expand_paths (os_data%whizard_models_libpath)
  call expand_paths (os_data%whizard_susypath)
  call expand_paths (os_data%whizard_gmlpath)
  call expand_paths (os_data%whizard_cutspath)
  call expand_paths (os_data%whizard_texpath)
  call expand_paths (os_data%whizard_testdatapath)
  call expand_paths (os_data%whizard_models_libpath_local)
  call expand_paths (os_data%whizard_modelpath_local)
  call expand_paths (os_data%whizard_omega_binpath_local)
  call expand_paths (os_data%latex)
  call expand_paths (os_data%gml)
  call expand_paths (os_data%dvips)
  call expand_paths (os_data%ps2pdf)
contains
  subroutine expand_paths (string)
    type(string_t), intent(inout) :: string
    integer :: i
    do i = N_VARIABLES, 1, -1
        string = replace (string, variable(i), value(i), every=.true.)
    end do
  end subroutine expand_paths
end subroutine os_data_expand_paths
```

Write contents

⟨*OS interface: public*⟩+≡

```
public :: os_data_write
```

⟨*OS interface: procedures*⟩+≡

```
subroutine os_data_write (os_data, unit)
  type(os_data_t), intent(in) :: os_data
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(A)")  "OS data:"
  write (u, *) "use_libtool   = ", os_data%use_libtool
  write (u, *) "use_testfiles = ", os_data%use_testfiles
  write (u, *) "fc            = ", char (os_data%fc)
  write (u, *) "fcflags       = ", char (os_data%fcflags)
```

```
    write (u, *) "fcflags_pic   = ", char (os_data%fcflags_pic)
    write (u, *) "fc_src_ext    = ", char (os_data%fc_src_ext)
    write (u, *) "obj_ext       = ", char (os_data%obj_ext)
    write (u, *) "ld            = ", char (os_data%ld)
    write (u, *) "ldflags       = ", char (os_data%ldflags)
    write (u, *) "ldflags_so    = ", char (os_data%ldflags_so)
    write (u, *) "ldflags_static = ", char (os_data%ldflags_static)
    write (u, *) "shlib_ext     = ", char (os_data%shlib_ext)
    write (u, *) "prefix        = ", char (os_data%prefix)
    write (u, *) "exec_prefix   = ", char (os_data%exec_prefix)
    write (u, *) "bindir        = ", char (os_data%bindir)
    write (u, *) "libdir        = ", char (os_data%libdir)
    write (u, *) "includedir    = ", char (os_data%includedir)
    write (u, *) "datarootdir   = ", char (os_data%datarootdir)
    write (u, *) "whizard_omega_binpath = ", &
         char (os_data%whizard_omega_binpath)
    write (u, *) "whizard_includes      = ", char (os_data%whizard_includes)
    write (u, *) "whizard_ldflags       = ", char (os_data%whizard_ldflags)
    write (u, *) "whizard_libtool       = ", char (os_data%whizard_libtool)
    write (u, *) "whizard_modelpath      = ", &
         char (os_data%whizard_modelpath)
    write (u, *) "whizard_models_libpath = ", &
         char (os_data%whizard_modelpath)
    write (u, *) "whizard_susypath      = ", char (os_data%whizard_includes)
    write (u, *) "whizard_gmlpath       = ", char (os_data%whizard_includes)
    write (u, *) "whizard_cutspath      = ", char (os_data%whizard_includes)
    write (u, *) "whizard_texpath       = ", char (os_data%whizard_includes)
    write (u, *) "whizard_testdatapath  = ", &
         char (os_data%whizard_testdatapath)
    write (u, *) "whizard_modelpath_local      = ", &
         char (os_data%whizard_modelpath_local)
    write (u, *) "whizard_models_libpath_local = ", &
         char (os_data%whizard_models_libpath_local)
    write (u, *) "whizard_omega_binpath_local  = ", &
         char (os_data%whizard_omega_binpath_local)
    write (u, *) "event_analysis_ps  = ", os_data%event_analysis_ps
    write (u, *) "event_analysis_pdf = ", os_data%event_analysis_pdf
    write (u, *) "latex  = ", char (os_data%latex)
    write (u, *) "gml    = ", char (os_data%gml)
    write (u, *) "dvips  = ", char (os_data%dvips)
    write (u, *) "ps2pdf = ", char (os_data%ps2pdf)
  end subroutine os_data_write
```

2.4.3 Dynamic linking

We define a type that holds the filehandle for a dynamically linked library
(shared object), together with functions to open and close the library, and to
access functions in this library.

⟨*OS interface: public*⟩+≡
```
    public :: dlaccess_t
```

⟨*OS interface: types*⟩+≡
```
    type :: dlaccess_t
```

```
      private
      type(string_t) :: filename
      type(c_ptr) :: handle = c_null_ptr
      logical :: is_open = .false.
      logical :: has_error = .false.
      type(string_t) :: error
   end type dlaccess_t
```

The interface to the library functions:

⟨*OS interface: interfaces*⟩≡
```
   interface
      function dlopen (filename, flag) result (handle) bind(C)
         import
         character(c_char), dimension(*) :: filename
         integer(c_int), value :: flag
         type(c_ptr) :: handle
      end function dlopen
   end interface

   interface
      function dlclose (handle) result (status) bind(C)
         import
         type(c_ptr), value :: handle
         integer(c_int) :: status
      end function dlclose
   end interface

   interface
      function dlerror () result (str) bind(C)
         import
         type(c_ptr) :: str
      end function dlerror
   end interface

   interface
      function dlsym (handle, symbol) result (fptr) bind(C)
         import
         type(c_ptr), value :: handle
         character(c_char), dimension(*) :: symbol
         type(c_funptr) :: fptr
      end function dlsym
   end interface
```

This reads an error string and transforms it into a `string_t` object, if an error has occured. If not, set the error flag to false and return an empty string.

⟨*Limits: public parameters*⟩+≡
```
   integer, parameter, public :: DLERROR_LEN = 160
```

⟨*OS interface: procedures*⟩+≡
```
   subroutine read_dlerror (has_error, error)
     logical, intent(out) :: has_error
     type(string_t), intent(out) :: error
     type(c_ptr) :: err_cptr
```

```
      character(len=DLERROR_LEN, kind=c_char), pointer :: err_fptr
      integer :: str_end
      err_cptr = dlerror ()
      if (c_associated (err_cptr)) then
         call c_f_pointer (err_cptr, err_fptr)
         has_error = .true.
         str_end = scan (err_fptr, c_null_char)
         if (str_end > 0) then
            error = err_fptr(1:str_end-1)
         else
            error = err_fptr
         end if
      else
         has_error = .false.
         error = ""
      end if
   end subroutine read_dlerror
```

This is the Fortran API. Init/final open and close the file, i.e., load and unload the library.

The flag with value 1 is defined as `RTLD_LAZY`, which means to resolve symbols only when they are used. The opposite would be `RTLD_NOW=2` (resolve everything immediately).

Note that a library can be opened more than once, and that for an ultimate close as many `dlclose` calls as `dlopen` calls are necessary. However, we assume that it is opened and closed only once.

⟨*OS interface: public*⟩+≡
```
   public :: dlaccess_init
   public :: dlaccess_final
```

⟨*OS interface: procedures*⟩+≡
```
   subroutine dlaccess_init (dlaccess, prefix, libname, os_data)
     type(dlaccess_t), intent(out) :: dlaccess
     type(string_t), intent(in) :: prefix, libname
     type(os_data_t), intent(in) :: os_data
     type(string_t) :: filename
     logical :: exist
     dlaccess%filename = libname
     filename = prefix // "/" // libname
     inquire (file=char(filename), exist=exist)
     if (.not. exist) then
        filename = prefix // "/.libs/" // libname
        inquire (file=char(filename), exist=exist)
        if (.not. exist) then
           dlaccess%has_error = .true.
           dlaccess%error = "Library '" // filename // "' not found"
           return
        end if
     end if
     dlaccess%handle = dlopen (char (filename) // c_null_char, 1_c_int)
     dlaccess%is_open = c_associated (dlaccess%handle)
     call read_dlerror (dlaccess%has_error, dlaccess%error)
   end subroutine dlaccess_init
```

```
subroutine dlaccess_final (dlaccess)
  type(dlaccess_t), intent(inout) :: dlaccess
  integer(c_int) :: status
  if (dlaccess%is_open) then
     status = dlclose (dlaccess%handle)
     dlaccess%is_open = .false.
     call read_dlerror (dlaccess%has_error, dlaccess%error)
  end if
end subroutine dlaccess_final
```

Return true if an error has occured.

⟨*OS interface: public*⟩+≡
```
public :: dlaccess_has_error
```

⟨*OS interface: procedures*⟩+≡
```
function dlaccess_has_error (dlaccess) result (flag)
  logical :: flag
  type(dlaccess_t), intent(in) :: dlaccess
  flag = dlaccess%has_error
end function dlaccess_has_error
```

Return the error string currently stored in the `dlaccess` object.

⟨*OS interface: public*⟩+≡
```
public :: dlaccess_get_error
```

⟨*OS interface: procedures*⟩+≡
```
function dlaccess_get_error (dlaccess) result (error)
  type(string_t) :: error
  type(dlaccess_t), intent(in) :: dlaccess
  error = dlaccess%error
end function dlaccess_get_error
```

The symbol handler returns the C address of the function with the given string
name. (It is a good idea to use `bind(C)` for all functions accessed by this, such
that the name string is well-defined.) Call `c_f_procpointer` to cast this into a
Fortran procedure pointer with an appropriate interface.

⟨*OS interface: public*⟩+≡
```
public :: dlaccess_get_c_funptr
```

⟨*OS interface: procedures*⟩+≡
```
function dlaccess_get_c_funptr (dlaccess, fname) result (fptr)
  type(c_funptr) :: fptr
  type(dlaccess_t), intent(inout) :: dlaccess
  type(string_t), intent(in) :: fname
  fptr = dlsym (dlaccess%handle, char (fname) // c_null_char)
  call read_dlerror (dlaccess%has_error, dlaccess%error)
end function dlaccess_get_c_funptr
```

2.4.4 Predicates

Return true if the library is loaded. In particular, this is false if loading was unsuccessful.

⟨*OS interface: public*⟩+≡
```
public :: dlaccess_is_open
```

⟨*OS interface: procedures*⟩+≡
```
function dlaccess_is_open (dlaccess) result (flag)
  logical :: flag
  type(dlaccess_t), intent(in) :: dlaccess
  flag = dlaccess%is_open
end function dlaccess_is_open
```

2.4.5 Shell access

This is the standard system call for executing a shell command, such as invoking a compiler.

In F2008 there will be the equivalent built-in command `execute_command_line`.

⟨*OS interface: public*⟩+≡
```
public :: os_system_call
```

⟨*OS interface: procedures*⟩+≡
```
subroutine os_system_call (command_string, status, verbose)
  type(string_t), intent(in) :: command_string
  integer, intent(out), optional :: status
  logical, intent(in), optional :: verbose
  logical :: verb
  integer :: stat
  verb = .false.;  if (present (verbose))  verb = verbose
  if (verb) &
      call msg_message ("command: " // char (command_string))
  stat = system (char (command_string) // c_null_char)
  if (present (status)) then
    status = stat
  else if (stat /= 0) then
    if (.not. verb) &
        call msg_message ("command: " // char (command_string))
    write (msg_buffer, "(A,I0)")  "Return code = ", stat
    call msg_message ()
    call msg_fatal ("System command returned with nonzero status code")
  end if
end subroutine os_system_call
```

⟨*OS interface: interfaces*⟩+≡
```
interface
  function system (command) result (status) bind(C)
    import
    integer(c_int) :: status
    character(c_char), dimension(*) :: command
  end function system
end interface
```

2.4.6 Querying for a directory

This queries for the existence of a directory. There is no standard way to achieve this in FORTRAN, and if we were to call into `libc`, we'd need access to C macros for evaluating the result, so we resort to calling `test` as a system call.

⟨*OS interface: public*⟩+≡
```
    public :: os_dir_exist
```

⟨*OS interface: procedures*⟩+≡
```
    function os_dir_exist (name) result (res)
      type(string_t), intent(in) :: name
      logical :: res
      integer :: status
      call os_system_call ('test -d "' // name // '"', status=status)
      res = status == 0
    end function os_dir_exist
```

2.4.7 Fortran compiler and linker

Compile a single module for use in a shared library, but without linking.

⟨*OS interface: public*⟩+≡
```
    public :: os_compile_shared
```

⟨*OS interface: procedures*⟩+≡
```
    subroutine os_compile_shared (src, os_data, status)
      type(string_t), intent(in) :: src
      type(os_data_t), intent(in) :: os_data
      integer, intent(out), optional :: status
      type(string_t) :: command_string
      if (os_data%use_libtool) then
         command_string = &
              os_data%whizard_libtool // " --mode=compile " // &
              os_data%fc // " " // &
              "-c " // &
              os_data%whizard_includes // " " // &
              os_data%fcflags // " " // &
              "'" // src // os_data%fc_src_ext // "'"
      else
         command_string = &
              os_data%fc // " " // &
              "-c  " // &
              os_data%fcflags_pic // " " // &
              os_data%whizard_includes // " " // &
              os_data%fcflags // " " // &
              "'" // src // os_data%fc_src_ext // "'"
      end if
      call os_system_call (command_string, status)
    end subroutine os_compile_shared
```

Link an array of object files to build a shared object library. In the libtool case, we have to specify a `-rpath`, otherwise only a static library can be built. However, since the library is never installed, this rpath is irrelevant.

⟨*OS interface: public*⟩+≡
```
    public :: os_link_shared
```

⟨*OS interface: procedures*⟩+≡
```
    subroutine os_link_shared (objlist, lib, os_data, status)
      type(string_t), intent(in) :: objlist, lib
      type(os_data_t), intent(in) :: os_data
      integer, intent(out), optional :: status
      type(string_t) :: command_string
      if (os_data%use_libtool) then
         command_string = &
              os_data%whizard_libtool // " --mode=link " // &
              os_data%fc // " " // &
              "-module " // &
              "-rpath /usr/local/lib" // " " // &
              os_data%fcflags // " " // &
              os_data%whizard_ldflags // " " // &
              os_data%ldflags // " " // &
              "-o '" // lib // ".la' " // &
              objlist
      else
         command_string = &
              os_data%ld // " " // &
              os_data%ldflags_so // " " // &
              os_data%fcflags // " " // &
              os_data%whizard_ldflags // " " // &
              os_data%ldflags // " " // &
              "-o '" // lib // os_data%shlib_ext // "' " // &
              objlist
      end if
      call os_system_call (command_string, status)
    end subroutine os_link_shared
```

Link an array of object files / libraries to build a static executable.

⟨*OS interface: public*⟩+≡
```
    public :: os_link_static
```

⟨*OS interface: procedures*⟩+≡
```
    subroutine os_link_static (objlist, exec_name, os_data, status)
      type(string_t), intent(in) :: objlist, exec_name
      type(os_data_t), intent(in) :: os_data
      integer, intent(out), optional :: status
      type(string_t) :: command_string
      if (os_data%use_libtool) then
         command_string = &
              os_data%whizard_libtool // " --mode=link " // &
              os_data%fc // " " // &
              "-static " // &
              os_data%whizard_ldflags // " " // &
              os_data%ldflags // " " // &
              os_data%ldflags_static // " " // &
              "-o '" // exec_name // "' " // &
              objlist
      else
         command_string = &
              os_data%ld // " " // &
              os_data%ldflags_so // " " // &
```

```
                    os_data%whizard_ldflags // " " // &
                    os_data%ldflags // " " // &
                    os_data%ldflags_static // " " // &
                    "-o '" // exec_name // "' " // &
                    objlist
           end if
        call os_system_call (command_string, status)
      end subroutine os_link_static
```

Determine the name of the shared library to link. If libtool is used, this is
encoded in the `.la` file which resides in place of the library itself.

⟨*OS interface: public*⟩+≡
```
      public :: os_get_dlname
```

⟨*OS interface: procedures*⟩+≡
```
      function os_get_dlname (lib, os_data, ignore, silent) result (dlname)
        type(string_t) :: dlname
        type(string_t), intent(in) :: lib
        type(os_data_t), intent(in) :: os_data
        logical, intent(in), optional :: ignore, silent
        type(string_t) :: filename
        type(string_t) :: buffer
        logical :: exist, required, quiet
        integer :: u
        u = free_unit ()
        if (present (ignore)) then
           required = .not. ignore
        else
           required = .true.
        end if
           if (present (silent)) then
           quiet = silent
        else
           quiet = .false.
        end if
        if (os_data%use_libtool) then
           filename = lib // ".la"
           inquire (file=char(filename), exist=exist)
           if (exist) then
              open (unit=u, file=char(filename), action="read", status="old")
              SCAN_LTFILE: do
                 call get (u, buffer)
                 if (extract (buffer, 1, 7) == "dlname=") then
                    dlname = extract (buffer, 9)
                    dlname = remove (dlname, len (dlname))
                    exit SCAN_LTFILE
                 end if
              end do SCAN_LTFILE
              close (u)
           else if (required) then
              if (.not. quiet) call msg_fatal (" Library '" // char (lib) &
                   // "': libtool archive not found")
              dlname = ""
           else
```

31

```
                 if (.not. quiet) call msg_message ("[No compiled library '" &
                      // char (lib) // "']")
                 dlname = ""
              end if
        else
           dlname = lib // os_data%shlib_ext
           inquire (file=char(dlname), exist=exist)
           if (.not. exist) then
              if (required) then
                 if (.not. quiet) call msg_fatal (" Library '" // char (lib) &
                      // "' not found")
              else
                 if (.not. quiet) call msg_message &
                      ("[No compiled process library '" // char (lib) // "']")
                 dlname = ""
              end if
           end if
        end if
     end function os_get_dlname
```

2.4.8 Test

⟨*OS interface: public*⟩+≡
```
     public :: os_interface_test
```

⟨*OS interface: procedures*⟩+≡
```
     subroutine os_interface_test ()
       call os_interface_test1 ()
     end subroutine os_interface_test
```

Write a Fortran source file, compile it to a shared library, load it, and execute
the contained function.

⟨*OS interface: procedures*⟩+≡
```
     subroutine os_interface_test1 ()
       type(dlaccess_t) :: dlaccess
       type(string_t) :: fname, libname
       type(os_data_t) :: os_data
       type(string_t) :: filename_src, filename_obj
       interface
          function so_test_proc (i) result (j) bind(C)
            import c_int
            integer(c_int), intent(in) :: i
            integer(c_int) :: j
          end function so_test_proc
       end interface
       procedure(so_test_proc), pointer :: so_test => null ()
       type(c_funptr) :: c_fptr
       integer :: u
       integer(c_int) :: i
       call os_data_init (os_data)
       fname = "so_test"
       filename_src = fname // os_data%fc_src_ext
```

```
      filename_obj = fname // os_data%obj_ext
      libname = fname // os_data%shlib_ext
      print *, "* write source file 'so_test.f90'"
      u = free_unit ()
      open (unit=u, file=char(filename_src), action="write")
      write (u, "(A)")  "function so_test (i) result (j) bind(C)"
      write (u, "(A)")  "  integer(c_int), intent(in) :: i"
      write (u, "(A)")  "  integer(c_int) :: j"
      write (u, "(A)")  "  j = 2 * i"
      write (u, "(A)")  "end function so_test"
      close (u)
      print *, "* compile and link as 'so_test.so'"
      call os_compile_shared (fname, os_data)
      call os_link_shared (filename_obj, fname, os_data)
      print *, "* load library 'so_test.so'"
      call dlaccess_init (dlaccess, var_str ("."), libname, os_data)
      if (dlaccess_is_open (dlaccess)) then
         print *, "  success"
      else
         print *, "  failure"
      end if
      print *, "* load symbol 'so_test'"
      c_fptr = dlaccess_get_c_funptr (dlaccess, fname)
      if (c_associated (c_fptr)) then
         print *, "  success"
      else
         print *, "  failure"
      end if
      call c_f_procpointer (c_fptr, so_test)
      print *, "* Execute function from 'so_test.so'"
      i = 7
      print *, "  input = ", i
      print *, "  result =", so_test(i)
      print *, "* Cleanup"
      call dlaccess_final (dlaccess)
    end subroutine os_interface_test1
```

2.5   Accessing the system clock

Fortran 90 provides a standard interface for the system clock. Here, we define
a record for storing the information, and higher-level functions for accessing it.

⟨clock.f90⟩≡
  ⟨*File header*⟩

  module clock

  ⟨*Use kinds*⟩
  ⟨*Use file utils*⟩

  ⟨*Standard module head*⟩

  ⟨*Clock: public*⟩

⟨*Clock: types*⟩

⟨*Clock: interfaces*⟩

```
contains
```

⟨*Clock: procedures*⟩

```
end module clock
```

The format for storing the current date. We want to be able to store time differences, therefore everything is converted to the time difference between now and Jan 01 2000, 0:00. The timezone is ignored.

⟨*Clock: public*⟩≡
```
    public :: time_t
```

⟨*Clock: types*⟩≡
```
    type :: time_t
       integer :: day = 0
       integer :: hour = 0
       integer :: minute = 0
       integer :: second = 0
       integer :: millisecond = 0
    end type time_t
```

Write the time in readable form

⟨*Clock: public*⟩+≡
```
    public :: time_write
```

⟨*Clock: interfaces*⟩≡
```
    interface time_write
       module procedure time_write_unit
       module procedure time_write_string
    end interface
```

The maximal time that can be written is 999 days. No way to write negative times.

⟨*Clock: procedures*⟩≡
```
    subroutine time_write_unit (t, unit, advance, days, seconds, milliseconds)
      type(time_t), intent(in) :: t
      integer, intent(in) :: unit
      character(len=*), intent(in), optional :: advance
      logical, intent(in), optional :: days, seconds, milliseconds
      logical :: dd, ss, ms
      character(len=3) :: adv
      if (present (advance)) then
         adv = advance
      else
         adv = "yes"
      end if
      dd = .false.;  if (present (days)) dd = days
      ss = .true.;   if (present (seconds))  ss = seconds
      ms = .false.;  if (present (milliseconds)) ms = milliseconds
      if (dd) then
         write (unit, "(I3,A,1x,I2.2,A,1x,I2.2,A)", advance="no") &
```

```
           t%day, "d", t%hour, "h", t%minute, "m"
      else
         write (unit, "(I5,A,1x,I2.2,A)", advance="no") &
               t%day * 24 + t%hour, "h", t%minute, "m"
      end if
      if (ss) then
         write (unit, "(1x,I2.2)", advance="no")  t%second
         if (ms) then
            write (unit, "(A,I3.3,A)", advance=trim(adv))  ".", t%millisecond, "s"
         else
            write (unit, "(A)", advance=trim(adv)) "s"
         end if
      end if
   end subroutine time_write_unit
```

Write the time to a string. This uses a scratch file for simplicity. Non-advancing output is not possible.

⟨*Clock: procedures*⟩+≡
```
   subroutine time_write_string (t, string, days, seconds, milliseconds)
     type(time_t), intent(in) :: t
     character(*), intent(out) :: string
     logical, intent(in), optional :: days, seconds, milliseconds
     integer :: unit
     unit = free_unit ()
     open (unit, status="scratch", action="readwrite")
     call time_write_unit (t, unit, &
          days=days, seconds=seconds, milliseconds=milliseconds)
     rewind (unit)
     read (unit, "(A)") string
     close (unit)
   end subroutine time_write_string
```

Set the current time to number of days, minutes, seconds, milliseconds elapsed since Jan 1 2000. Works for dates after Jan 1 2001 until Dec 31 2099.

⟨*Clock: public*⟩+≡
```
   public :: time_current
```

⟨*Clock: procedures*⟩+≡
```
   function time_current () result (t)
     type(time_t) :: t
     integer, dimension(8) :: values
     integer :: year
     call date_and_time (values = values)
     t%millisecond = values(8)
     t%second = values(7)
     t%minute = values(6)
     t%hour = values(5)
     ! values(4) is the difference local time - GMT in minutes
     t%day = values(3) - 1
     select case (values(2))
     case ( 1)
     case ( 2); t%day = t%day + 31
     case ( 3); t%day = t%day + 31 + 28
```

```
      case ( 4); t%day = t%day + 31 + 28 + 31
      case ( 5); t%day = t%day + 31 + 28 + 31 + 30
      case ( 6); t%day = t%day + 31 + 28 + 31 + 30 + 31
      case ( 7); t%day = t%day + 31 + 28 + 31 + 30 + 31 + 30
      case ( 8); t%day = t%day + 31 + 28 + 31 + 30 + 31 + 30 + 31
      case ( 9); t%day = t%day + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31
      case (10); t%day = t%day + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30
      case (11); t%day = t%day + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31
      case (12); t%day = t%day + 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 30
      end select
      year = values(1) - 2000
      if (year < 1)  return
      t%day = t%day + 365 * year + ((year-1) / 4 + 1)
      if (mod (year, 4) == 0 .and. values(2) > 2)  t%day = t%day + 1
    end function time_current
```

Convert from and to seconds

⟨*Clock: public*⟩+≡
```
    public :: time_to_seconds, time_from_seconds
```

⟨*Clock: procedures*⟩+≡
```
    function time_to_seconds (t) result (s)
      type(time_t), intent(in) :: t
      real(kind=default) :: s
      s = t%millisecond / 1000._default &
           + t%second &
           + 60._default * (t%minute &
                            + 60._default * (t%hour &
                                             + 24._default * t%day))
    end function time_to_seconds
```

⟨*Clock: procedures*⟩+≡
```
    function time_from_seconds (s) result (t)
      real(kind=default), intent(in) :: s
      type(time_t) :: t
      t%millisecond = s * 1000
      t%second = t%millisecond / 1000
      t%millisecond = mod (t%millisecond, 1000)
      t%minute = t%second / 60
      t%second = mod (t%second, 60)
      t%hour = t%minute / 60
      t%minute = mod (t%minute, 60)
      t%day = t%hour / 24
      t%hour = mod (t%hour, 24)
    end function time_from_seconds
```

Add and subtract times

⟨*Clock: public*⟩+≡
```
    public :: operator(+), operator(-)
```

⟨*Clock: interfaces*⟩+≡
```
    interface operator(+)
       module procedure time_add
```

```
      end interface
      interface operator(-)
         module procedure time_subtract
      end interface
```
⟨*Clock: procedures*⟩+≡
```
      function time_add (t1, t2) result (t)
        type(time_t), intent(in) :: t1, t2
        type(time_t) :: t
        t%millisecond = t1%millisecond + t2%millisecond
        t%second = t1%second + t2%second
        t%minute = t1%minute + t2%minute
        t%hour = t1%hour + t2%hour
        t%day = t1%day + t2%day
        if (t%millisecond > 999) then
           t%second = t%second + t%millisecond / 1000
           t%millisecond = modulo (t%millisecond, 1000)
        end if
        if (t%second > 59) then
           t%minute = t%minute + t%second / 60
           t%second = modulo (t%second, 60)
        end if
        if (t%minute > 59) then
           t%hour = t%hour + t%minute / 60
           t%minute = modulo (t%minute, 60)
        end if
        if (t%hour > 23) then
           t%day = t%day + t%hour / 24
           t%hour = modulo (t%hour, 24)
        end if
      end function time_add
```

This works for positive difference only, the first time must be later than the
second. Otherwise, the number of days is set to zero.

⟨*Clock: procedures*⟩+≡
```
      function time_subtract (t1, t2) result (t)
        type(time_t), intent(in) :: t1, t2
        type(time_t) :: t
        t%millisecond = t1%millisecond - t2%millisecond
        t%second = t1%second - t2%second
        t%minute = t1%minute - t2%minute
        t%hour = t1%hour - t2%hour
        t%day = t1%day - t2%day
        if (t%millisecond < 0) then
           t%second = t%second - 1 + t%millisecond / 1000
           t%millisecond = modulo (t%millisecond, 1000)
        end if
        if (t%second < 0) then
           t%minute = t%minute - 1 + t%second / 60
           t%second = modulo (t%second, 60)
        end if
        if (t%minute < 0) then
           t%hour = t%hour - 1 + t%minute / 60
           t%minute = modulo (t%minute, 60)
```

```
        end if
        if (t%hour < 0) then
            t%day = t%day - 1 + t%hour / 24
            t%hour = modulo (t%hour, 24)
        end if
        if (t%day < 0) then
            t%day = 0
        end if
    end function time_subtract
```

Compare times

⟨*Clock: public*⟩+≡
```
    public :: operator(==)
    public :: operator(<), operator(>)
    public :: operator(<=), operator(>=)
```

⟨*Clock: interfaces*⟩+≡
```
    interface operator(==)
      module procedure time_equal
    end interface
    interface operator(<)
      module procedure time_less_than
    end interface
    interface operator(>)
      module procedure time_greater_than
    end interface
    interface operator(<=)
      module procedure time_less_equal
    end interface
    interface operator(>=)
      module procedure time_greater_equal
    end interface
```

⟨*Clock: procedures*⟩+≡
```
    function time_equal (t1, t2) result (equal)
      type(time_t), intent(in) :: t1, t2
      logical :: equal
      equal = (t1%day==t2%day) .and. (t1%hour==t2%hour) .and. &
            (t1%minute==t2%minute) .and. (t1%second==t2%second) .and. &
            (t1%millisecond==t2%millisecond)
    end function time_equal
```

⟨*Clock: procedures*⟩+≡
```
    function time_less_than (t1, t2) result (less)
      type(time_t), intent(in) :: t1, t2
      logical :: less
      if (t1%day < t2%day) then
         less = .true.
      else if (t1%day == t2%day) then
         if (t1%hour < t2%hour) then
            less = .true.
         else if (t1%hour == t2%hour) then
            if (t1%minute < t2%minute) then
               less = .true.
```

```
            else if (t1%minute == t2%minute) then
                if (t1%second < t2%second) then
                    less = .true.
                else if (t1%second == t2%second) then
                    if (t1%millisecond < t2%millisecond) then
                        less = .true.
                    else
                        less = .false.
                    end if
                else
                    less = .false.
                end if
            else
                less = .false.
            end if
        else
            less = .false.
        end if
    end function time_less_than
```

⟨*Clock: procedures*⟩+≡
```
    function time_greater_than (t1, t2) result (greater)
      type(time_t), intent(in) :: t1, t2
      logical :: greater
      greater = time_less_than (t2, t1)
    end function time_greater_than
```

⟨*Clock: procedures*⟩+≡
```
    function time_less_equal (t1, t2) result (less_equal)
      type(time_t), intent(in) :: t1, t2
      logical :: less_equal
      less_equal = time_equal (t1, t2) .or. time_less_than (t1, t2)
    end function time_less_equal
```

⟨*Clock: procedures*⟩+≡
```
    function time_greater_equal (t1, t2) result (greater_equal)
      type(time_t), intent(in) :: t1, t2
      logical :: greater_equal
      greater_equal = time_equal (t1, t2) .or. time_greater_than (t1, t2)
    end function time_greater_equal
```

2.6   Hashtables

Hash tables, like lists, are not part of Fortran and must be defined on a per-case
basis. In this section we define a module that contains a hash function.

Furthermore, for reference there is a complete framework of hashtable type
definitions and access functions. This code is to be replicated where hash tables

are used, mutatis mutandis.

⟨`hashes.f90`⟩≡
  ⟨*File header*⟩

  ```
module hashes
```

    ```
use kinds, only: i8, i32 !NODEP!
```

  ⟨*Standard module head*⟩

  ⟨*Hashes: public*⟩

  ```
contains
```

  ⟨*Hashes: procedures*⟩

  ```
end module hashes
```

2.6.1 The hash function

This is the one-at-a-time hash function by Bob Jenkins (from Wikipedia), re-implemented in Fortran. The function works on an array of bytes (8-bit integers), as could be produced by, e.g., the `transfer` function, and returns a single 32-bit integer. For determining the position in a hashtable, one can pick the lower bits of the result as appropriate to the hashtable size (which should be a power of 2). Note that we are working on signed integers, so the interpretation of values differs from the C version. This should not matter in practice, however.

⟨*Hashes: public*⟩≡
```
    public :: hash
```

⟨*Hashes: procedures*⟩≡
```
    function hash (key)
      integer(i8), dimension(:), intent(in) :: key
      integer(i32) :: hash
      integer :: i
      hash = 0
      do i = 1, size (key)
         hash = hash + key(i)
         hash = hash + ishft (hash, 10)
         hash = ieor (hash, ishft (hash, -6))
      end do
      hash = hash + ishft (hash, 3)
      hash = ieor (hash, ishft (hash, -11))
      hash = hash + ishft (hash, 15)
    end function hash
```

2.6.2 The hash table

We define a generic hashtable type (that depends on the `hash_data_t` type) together with associated methods.

This is a template:

⟨*Hashtables: types*⟩≡
```
type :: hash_data_t
   integer :: i
end type hash_data_t
```

Associated methods:

⟨*Hashtables: procedures*⟩≡
```
subroutine hash_data_final (data)
  type(hash_data_t), intent(inout) :: data
end subroutine hash_data_final

subroutine hash_data_write (data, unit)
  type(hash_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  write (u, *) data%i
end subroutine hash_data_write
```

Each hash entry stores the unmasked hash value, the key, and points to actual data if present. Note that this could be an allocatable scalar in principle, but making it a pointer avoids deep copy when expanding the hashtable.

⟨*Hashtables: types*⟩+≡
```
type :: hash_entry_t
   integer(i32) :: hashval = 0
   integer(i8), dimension(:), allocatable :: key
   type(hash_data_t), pointer :: data => null ()
end type hash_entry_t
```

The hashtable object holds the actual table, the number of filled entries and the number of entries after which the size should be doubled. The mask is equal to the table size minus one and thus coincides with the upper bound of the table index, which starts at zero.

⟨*Hashtables: types*⟩+≡
```
type :: hashtable_t
   integer :: n_entries = 0
   real :: fill_ratio = 0
   integer :: n_entries_max = 0
   integer(i32) :: mask = 0
   type(hash_entry_t), dimension(:), allocatable :: entry
end type hashtable_t
```

Initializer: The size has to be a power of two, the fill ratio is a real (machine-default!) number between 0 and 1.

⟨*Hashtables: procedures*⟩+≡
```
subroutine hashtable_init (hashtable, size, fill_ratio)
  type(hashtable_t), intent(out) :: hashtable
  integer, intent(in) :: size
  real, intent(in) :: fill_ratio
  hashtable%fill_ratio = fill_ratio
```

```
      hashtable%n_entries_max = size * fill_ratio
      hashtable%mask = size - 1
      allocate (hashtable%entry (0:hashtable%mask))
   end subroutine hashtable_init
```

Finalizer: This calls a `hash_data_final` subroutine which must exist.

⟨*Hashtables: procedures*⟩+≡
```
   subroutine hashtable_final (hashtable)
     type(hashtable_t), intent(inout) :: hashtable
     integer :: i
     do i = 0, hashtable%mask
        if (associated (hashtable%entry(i)%data)) then
           call hash_data_final (hashtable%entry(i)%data)
           deallocate (hashtable%entry(i)%data)
        end if
     end do
     deallocate (hashtable%entry)
   end subroutine hashtable_final
```

Output. Here, we refer to a `hash_data_write` subroutine.

⟨*Hashtables: procedures*⟩+≡
```
   subroutine hashtable_write (hashtable, unit)
     type(hashtable_t), intent(in) :: hashtable
     integer, intent(in), optional :: unit
     integer :: u, i
     u = output_unit (unit);  if (u < 0)  return
     do i = 0, hashtable%mask
        if (associated (hashtable%entry(i)%data)) then
           write (u, *) i, "(hash =", hashtable%entry(i)%hashval, ")", &
                hashtable%entry(i)%key
           call hash_data_write (hashtable%entry(i)%data, unit)
        end if
     end do
   end subroutine hashtable_write
```

2.6.3   Hashtable insertion

Insert a single entry with the hash value as trial place. If the table is filled, first expand it.

⟨*Hashtables: procedures*⟩+≡
```
   subroutine hashtable_insert (hashtable, key, data)
     type(hashtable_t), intent(inout) :: hashtable
     integer(i8), dimension(:), intent(in) :: key
     type(hash_data_t), intent(in), target :: data
     integer(i32) :: h
     if (hashtable%n_entries >= hashtable%n_entries_max) &
          call hashtable_expand (hashtable)
     h = hash (key)
     call hashtable_insert_rec (hashtable, h, h, key, data)
   end subroutine hashtable_insert
```

We need this auxiliary routine for doubling the size of the hashtable. We rely on
the fact that default assignment copies the data pointer, not the data themselves.
The temporary array must not be finalized; it is deallocated automatically to-
gether with its allocatable components.

⟨*Hashtables: procedures*⟩+≡

```
subroutine hashtable_expand (hashtable)
  type(hashtable_t), intent(inout) :: hashtable
  type(hash_entry_t), dimension(:), allocatable :: table_tmp
  integer :: i, s
  allocate (table_tmp (0:hashtable%mask))
  table_tmp = hashtable%entry
  deallocate (hashtable%entry)
  s = 2 * size (table_tmp)
  hashtable%n_entries = 0
  hashtable%n_entries_max = s * hashtable%fill_ratio
  hashtable%mask = s - 1
  allocate (hashtable%entry (0:hashtable%mask))
  do i = 0, ubound (table_tmp, 1)
     if (associated (table_tmp(i)%data)) then
        call hashtable_insert_rec (hashtable, table_tmp(i)%hashval, &
             table_tmp(i)%hashval, table_tmp(i)%key, table_tmp(i)%data)
     end if
  end do
end subroutine hashtable_expand
```

Insert a single entry at a trial place h, reduced to the table size. Collision
resolution is done simply by choosing the next element, recursively until the
place is empty. For bookkeeping, we preserve the original hash value. For a
good hash function, there should be no clustering.

Note that if the new key exactly matches an existing key, nothing is done.

⟨*Hashtables: procedures*⟩+≡

```
recursive subroutine hashtable_insert_rec (hashtable, h, hashval, key, data)
  type(hashtable_t), intent(inout) :: hashtable
  integer(i32), intent(in) :: h, hashval
  integer(i8), dimension(:), intent(in) :: key
  type(hash_data_t), intent(in), target :: data
  integer(i32) :: i
  i = iand (h, hashtable%mask)
  if (associated (hashtable%entry(i)%data)) then
     if (size (hashtable%entry(i)%key) /= size (key)) then
        call hashtable_insert_rec (hashtable, h + 1, hashval, key, data)
     else if (any (hashtable%entry(i)%key /= key)) then
        call hashtable_insert_rec (hashtable, h + 1, hashval, key, data)
     end if
  else
     hashtable%entry(i)%hashval = hashval
     allocate (hashtable%entry(i)%key (size (key)))
     hashtable%entry(i)%key = key
     hashtable%entry(i)%data => data
     hashtable%n_entries = hashtable%n_entries + 1
  end if
end subroutine hashtable_insert_rec
```

2.6.4 Hashtable lookup

The lookup function has to parallel the insert function. If the place is filled,
check if the key matches. Yes: return the pointer; no: increment the hash value
and check again.

⟨*Hashtables: procedures*⟩+≡
```
    function hashtable_lookup (hashtable, key) result (ptr)
      type(hash_data_t), pointer :: ptr
      type(hashtable_t), intent(in) :: hashtable
      integer(i8), dimension(:), intent(in) :: key
      ptr => hashtable_lookup_rec (hashtable, hash (key), key)
    end function hashtable_lookup
```

⟨*Hashtables: procedures*⟩+≡
```
    recursive function hashtable_lookup_rec (hashtable, h, key) result (ptr)
      type(hash_data_t), pointer :: ptr
      type(hashtable_t), intent(in) :: hashtable
      integer(i32), intent(in) :: h
      integer(i8), dimension(:), intent(in) :: key
      integer(i32) :: i
      i = iand (h, hashtable%mask)
      if (associated (hashtable%entry(i)%data)) then
         if (size (hashtable%entry(i)%key) == size (key)) then
            if (all (hashtable%entry(i)%key == key)) then
               ptr => hashtable%entry(i)%data
            else
               ptr => hashtable_lookup_rec (hashtable, h + 1, key)
            end if
         else
            ptr => hashtable_lookup_rec (hashtable, h + 1, key)
         end if
      else
         ptr => null ()
      end if
    end function hashtable_lookup_rec
```

⟨*Hashtables: public*⟩≡
```
    public :: hashtable_test
```

⟨*Hashtables: procedures*⟩+≡
```
    subroutine hashtable_test ()
      type(hash_data_t), pointer :: data
      type(hashtable_t) :: hashtable
      integer(i8) :: i
      call hashtable_init (hashtable, 16, 0.25)
      do i = 1, 10
         allocate (data)
         data%i = i*i
         call hashtable_insert (hashtable, (/i, i+i/), data)
      end do
      call hashtable_insert (hashtable, (/2_i8, 4_i8/), data)
      call hashtable_write (hashtable)
      data => hashtable_lookup (hashtable, (/5_i8, 10_i8/))
      if (associated (data)) then
```

```
      print *, "lookup:", data%i
    else
      print *, "lookup: --"
    end if
    data => hashtable_lookup (hashtable, (/6_i8, 12_i8/))
    if (associated (data)) then
      print *, "lookup:", data%i
    else
      print *, "lookup: --"
    end if
    data => hashtable_lookup (hashtable, (/4_i8, 9_i8/))
    if (associated (data)) then
      print *, "lookup:", data%i
    else
      print *, "lookup: --"
    end if
    call hashtable_final (hashtable)
  end subroutine hashtable_test
```

2.7   Message handling

We are not so ambitious as to do proper exception handling in WHIZARD, but
at least it may be useful to have a common interface for diagnostics: Results,
messages, warnings, and such. As module variables we keep a buffer where
the current message may be written to and a level indicator which tells which
messages should be written on screen and which ones should be skipped. Alter-
natively, a string may be directly supplied to the message routine: this overrides
the buffer, avoiding the necessety of formatted I/O in trivial cases.

⟨diagnostics.f90⟩≡
  ⟨File header⟩

  module diagnostics

    use iso_c_binding !NODEP!
    use system_dependencies !NODEP!
    use kinds, only: i64 !NODEP!
  ⟨Use strings⟩
    use limits, only: BUFFER_SIZE, MAX_ERRORS !NODEP!
    use file_utils !NODEP!

  ⟨Standard module head⟩

  ⟨Diagnostics: public⟩

  ⟨Diagnostics: parameters⟩

  ⟨Diagnostics: types⟩

  ⟨Diagnostics: variables⟩

  ⟨Diagnostics: interfaces⟩

```
    contains
```

⟨*Diagnostics: procedures*⟩

```
    end module diagnostics
```

Diagnostics levels:

⟨*Diagnostics: parameters*⟩≡
```
    integer, parameter :: &
        & TERMINATE=-2, BUG=-1, &
        & FATAL=1, ERROR=2, WARNING=3, MESSAGE=4, RESULT=5, DEBUG=6
```

⟨*Diagnostics: variables*⟩≡
```
    integer, save :: msg_level = RESULT
```

Mask fatal errors so that are treated as normal errors. Useful for interactive mode.

⟨*Diagnostics: public*⟩≡
```
    public :: mask_fatal_errors
```

⟨*Diagnostics: variables*⟩+≡
```
    logical, save :: mask_fatal_errors = .false.
```

How to handle bugs and unmasked fatal errors. Either execute a normal stop statement, or call the C `exit()` function, or try to cause a program crash by dereferencing a null pointer.

⟨*Diagnostics: parameters*⟩+≡
```
    integer, parameter :: TERM_STOP = 0, TERM_EXIT = 1, TERM_CRASH = 2
```

⟨*Diagnostics: variables*⟩+≡
```
    integer, save :: handle_fatal_errors = TERM_EXIT
```

Keep track of errors. This might be used for exception handling, later. The counter is incremented only for screen messages, to avoid double counting.

⟨*Diagnostics: public*⟩+≡
```
    public :: msg_count
```

⟨*Diagnostics: variables*⟩+≡
```
    integer, dimension(TERMINATE:DEBUG), save :: msg_count = 0
```

Keep a list of all errors and warnings. Since we do not know the number of entries beforehand, we use a linked list.

⟨*Diagnostics: types*⟩≡
```
    type :: string_list
       character(len=BUFFER_SIZE) :: string
       type(string_list), pointer :: next
    end type string_list
    type :: string_list_pointer
       type(string_list), pointer :: first, last
    end type string_list_pointer
```

⟨*Diagnostics: variables*⟩+≡
```
    type(string_list_pointer), dimension(TERMINATE:WARNING), save :: &
        & msg_list = string_list_pointer (null(), null())
```

Add the current message buffer contents to the internal list.

⟨Diagnostics: procedures⟩≡
```
subroutine msg_add (level)
  integer, intent(in) :: level
  type(string_list), pointer :: message
  select case (level)
  case (TERMINATE:WARNING)
     allocate (message)
     message%string = msg_buffer
     nullify (message%next)
     if (.not.associated (msg_list(level)%first)) &
           & msg_list(level)%first => message
     if (associated (msg_list(level)%last)) &
           & msg_list(level)%last%next => message
     msg_list(level)%last => message
     msg_count(level) = msg_count(level) + 1
  end select
end subroutine msg_add
```

Initialization:

⟨Diagnostics: public⟩+≡
```
public :: msg_list_clear
```

⟨Diagnostics: procedures⟩+≡
```
subroutine msg_list_clear
  integer :: level
  type(string_list), pointer :: message
  do level = TERMINATE, WARNING
     do while (associated (msg_list(level)%first))
        message => msg_list(level)%first
        msg_list(level)%first => message%next
        deallocate (message)
     end do
     nullify (msg_list(level)%last)
  end do
  msg_count = 0
end subroutine msg_list_clear
```

Display the summary of errors and warnings (no need to count fatals. . .)

⟨Diagnostics: public⟩+≡
```
public :: msg_summary
```

⟨Diagnostics: procedures⟩+≡
```
   subroutine msg_summary (unit)
     integer, intent(in), optional :: unit
     call expect_summary (unit)
 1   format (A,1x,I2,1x,A,I2,1x,A)
     if (msg_count(ERROR) > 0 .and. msg_count(WARNING) > 0) then
        write (msg_buffer, 1) "There were", &
              & msg_count(ERROR), "error(s) and  ", &
              & msg_count(WARNING), "warning(s)."
        call msg_message (unit=unit)
     else if (msg_count(ERROR) > 0) then
        write (msg_buffer, 1) "There were", &
```

```
           & msg_count(ERROR), "error(s) and no warnings."
        call msg_message (unit=unit)
     else if (msg_count(WARNING) > 0) then
        write (msg_buffer, 1) "There were no errors and  ", &
              & msg_count(WARNING), "warning(s)."
        call msg_message (unit=unit)
     end if
  end subroutine msg_summary
```

Print the list of all messages of a given level.

⟨*Diagnostics: public*⟩+≡
```
    public :: msg_listing
```

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine msg_listing (level, unit, prefix)
      integer, intent(in) :: level
      integer, intent(in), optional :: unit
      character(len=*), intent(in), optional :: prefix
      type(string_list), pointer :: message
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      if (present (unit))  u = unit
      message => msg_list(level)%first
      do while (associated (message))
         if (present (prefix)) then
            write (u, "(A)") prefix // trim (message%string)
         else
            write (u, "(A)") trim (message%string)
         end if
         message => message%next
      end do
    end subroutine msg_listing
```

There is a hard limit on the line length which we should export. This buffer size is used both by the message handler and by the lexer below.

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: BUFFER_SIZE = 1000
```

The message buffer:

⟨*Diagnostics: public*⟩+≡
```
    public :: msg_buffer
```

⟨*Diagnostics: variables*⟩+≡
```
    character(len=BUFFER_SIZE), save :: msg_buffer = " "
```

After a message is issued, the buffer should be cleared:

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine buffer_clear
      msg_buffer = " "
    end subroutine buffer_clear
```

The generic handler for messages. If the unit is omitted (or = 6), the message is written to standard output if the precedence if sufficiently high (as determined by the value of `msg_level`). If the string is omitted, the buffer is used. In any case, the buffer is cleared after printing. In accordance with FORTRAN custom, the first column in the output is left blank. For messages and warnings, an additional exclamation mark and a blank is prepended. Furthermore, each message is appended to the internal message list (without prepending anything).

⟨*Diagnostics: procedures*⟩+≡

```
subroutine message_print (level, string, str_arr, unit, logfile)
    integer, intent(in) :: level
    character(len=*), intent(in), optional :: string
    type(string_t), dimension(:), intent(in), optional :: str_arr
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: logfile
    type(string_t) :: prep_string, aux_string, head_footer
    integer :: lu, i
    logical :: severe, is_error
    severe = .false.
    head_footer = "*******************************************************************************
    aux_string = ""
    is_error = .false.
!   integer :: proc_id
!   call mpi90_rank (proc_id)
!   if (proc_id==WHIZARD_ROOT) then
      select case (level)
      case (TERMINATE)
         prep_string = ""
      case (BUG)
         prep_string  = "*** WHIZARD BUG: "
         aux_string   = "***              "
         severe = .true.
         is_error = .true.
      case (FATAL)
         prep_string  = "*** FATAL ERROR: "
         aux_string   = "***              "
         severe = .true.
         is_error = .true.
      case (ERROR)
         prep_string = "*** ERROR: "
         aux_string  = "***        "
         is_error = .true.
      case (WARNING)
         prep_string = "Warning: "
      case (MESSAGE, DEBUG)
         prep_string = "| "
      case default
         prep_string = ""
      end select
      if (present(string))  msg_buffer = string
      lu = log_unit
      if (present(unit)) then
         if (unit/=6) then
            if (severe) write (unit, "(A)") char(head_footer)
```

49

```fortran
            if (is_error) write (unit, "(A)") char(head_footer)
            write (unit, "(A,A)") char(prep_string), trim(msg_buffer)
            if (present (str_arr)) then
               do i = 1, size(str_arr)
                  write (unit, "(A,A)") char(aux_string), char(trim(str_arr(i)))
               end do
            end if
            if (is_error) write (unit, "(A)") char(head_footer)
            if (severe) write (unit, "(A)") char(head_footer)
            lu = -1
         else if (level <= msg_level) then
            if (severe) print "(A)", char(head_footer)
            if (is_error) print "(A)", char(head_footer)
            print "(A,A)", char(prep_string), trim(msg_buffer)
            if (present (str_arr)) then
               do i = 1, size(str_arr)
                  print "(A,A)", char(aux_string), char(trim(str_arr(i)))
               end do
            end if
            if (is_error) print "(A)", char(head_footer)
            if (severe) print "(A)", char(head_footer)
            if (unit == log_unit)  lu = -1
         end if
      else if (level <= msg_level) then
         if (severe) print "(A)", char(head_footer)
         if (is_error) print "(A)", char(head_footer)
         print "(A,A)", char(prep_string), trim(msg_buffer)
            if (present (str_arr)) then
               do i = 1, size(str_arr)
                  print "(A,A)", char(aux_string), char(trim(str_arr(i)))
               end do
            end if
         if (is_error) print "(A)", char(head_footer)
         if (severe) print "(A)", char(head_footer)
      end if
      if (present (logfile)) then
         if (.not. logfile)  lu = -1
      end if
      if (lu >= 0) then
         if (severe) write (lu, "(A)") char(head_footer)
         if (is_error) write (lu, "(A)") char(head_footer)
         write (lu, "(A,A)")  char(prep_string), trim(msg_buffer)
         if (present (str_arr)) then
            do i = 1, size(str_arr)
               write (lu, "(A,A)") char(aux_string), char(trim(str_arr(i)))
            end do
         end if
         if (is_error) write (lu, "(A)") char(head_footer)
         if (severe) write (lu, "(A)") char(head_footer)
         flush (lu)
      end if
!    end if
   call msg_add (level)
   call buffer_clear
```

```
      end subroutine message_print
```

The specific handlers. In the case of fatal errors, bugs (failed assertions) and normal termination execution is stopped. For non-fatal errors a message is printed to standard output if no unit is given. Only if the number of `MAX_ERRORS` errors is reached, we abort the program. There are no further actions in the other cases, but this may change.

⟨*Diagnostics: public*⟩+≡
```
    public :: msg_terminate
    public :: msg_bug, msg_fatal, msg_error, msg_warning
    public :: msg_message, msg_result, msg_debug
```

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine msg_terminate (string, unit, quit_code)
      integer, intent(in), optional :: unit
      character(len=*), intent(in), optional :: string
      integer, intent(in), optional :: quit_code
      integer(c_int) :: return_code
      if (present (quit_code)) then
         return_code = quit_code
      else
         return_code = 0
      end if
      call msg_summary (unit)
      if (return_code == 0 .and. expect_failures /= 0) then
         return_code = 5
         call message_print (MESSAGE, &
              "WHIZARD run finished with 'expect' failure(s).", unit=unit)
      else
         call message_print (MESSAGE, "WHIZARD run finished.", unit=unit)
      end if
      call message_print (0, &
           "|=============================================================================|", unit=u
      call logfile_final ()
      if (return_code /= 0) then
         call exit (return_code)
      else
         stop
      end if
    end subroutine msg_terminate

    subroutine msg_bug (string, arr, unit)
      integer, intent(in), optional :: unit
      character(len=*), intent(in), optional :: string
      type(string_t), dimension(:), intent(in), optional :: arr
      logical, pointer :: crash_ptr => null ()
      call message_print (BUG, string, arr, unit)
      call msg_summary (unit)
      select case (handle_fatal_errors)
      case (TERM_EXIT)
         call message_print (TERMINATE, "WHIZARD run aborted.", unit=unit)
         ! call flush_all ()
         call exit (-1_c_int)
      case (TERM_CRASH)
```

```
      print *, "*** Intentional crash ***"
      print *, crash_ptr
    end select
    stop "WHIZARD run aborted."
  end subroutine msg_bug

  recursive subroutine msg_fatal (string, arr, unit)
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string
    type(string_t), dimension(:), intent(in), optional :: arr
    logical, pointer :: crash_ptr => null ()
    if (mask_fatal_errors) then
       call msg_error (string, arr, unit)
    else
       call message_print (FATAL, string, arr, unit)
       call msg_summary (unit)
       select case (handle_fatal_errors)
       case (TERM_EXIT)
          call message_print (TERMINATE, "WHIZARD run aborted.", unit=unit)
          ! call flush_all ()
          call exit (1_c_int)
       case (TERM_CRASH)
          print *, "*** Intentional crash ***"
          print *, crash_ptr
       end select
       stop "WHIZARD run aborted."
    end if
  end subroutine msg_fatal

  subroutine msg_error (string, arr, unit)
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string
    type(string_t), dimension(:), intent(in), optional :: arr
    call message_print (ERROR, string, arr, unit)
    if (msg_count(ERROR) >= MAX_ERRORS) then
       mask_fatal_errors = .false.
       call msg_fatal (" Too many errors encountered.")
!    else if (.not.present(unit) .and. .not.mask_fatal_errors)  then
!        call message_print (MESSAGE, "              (WHIZARD run continues)")
    end if
  end subroutine msg_error

  subroutine msg_warning (string, arr, unit)
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string
    type(string_t), dimension(:), intent(in), optional :: arr
    call message_print (WARNING, string, arr, unit)
  end subroutine msg_warning

  subroutine msg_message (string, unit, arr, logfile)
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string
    type(string_t), dimension(:), intent(in), optional :: arr
    logical, intent(in), optional :: logfile
```

```
      call message_print (MESSAGE, string, arr, unit, logfile)
    end subroutine msg_message

    subroutine msg_result (string, arr, unit, logfile)
      integer, intent(in), optional :: unit
      character(len=*), intent(in), optional :: string
      type(string_t), dimension(:), intent(in), optional :: arr
      logical, intent(in), optional :: logfile
      call message_print (RESULT, string, arr, unit, logfile)
    end subroutine msg_result

    subroutine msg_debug (string, arr, unit)
      integer, intent(in), optional :: unit
      character(len=*), intent(in), optional :: string
      type(string_t), dimension(:), intent(in), optional :: arr
      call message_print (DEBUG, string, arr, unit)
    end subroutine msg_debug
```

Interface to the standard clib exit function

⟨*Diagnostics: interfaces*⟩≡
```
    interface
       subroutine exit (status) bind (C)
         use iso_c_binding !NODEP!
         integer(c_int), value :: status
       end subroutine exit
    end interface
```

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: MAX_ERRORS = 10
```

Print the WHIZARD banner:

⟨*Diagnostics: public*⟩+≡
```
    public :: msg_banner
```

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine msg_banner (unit)
      integer, intent(in), optional :: unit
      call message_print (0, "|=================================================================
      call message_print (0, "|
      call message_print (0, "|    WW               WW WW   WW WW WWWWW     WW     WWWWW     WWWW
      call message_print (0, "|     WW    WW      WW   WW   WW WW    WW      WWWW    WW WW    WW
      call message_print (0, "|     WW WW WW WW     WWWWWWW WW    WW      WW WW    WWWWW     WW
      call message_print (0, "|      WWWW    WWWW      WW WW WW   WW     WWWWWWWW   WW WW    WW
      call message_print (0, "|       WW      WW      WW WW WW WWWWW WW      WW   WW   WW  WWWW
      call message_print (0, "|
      call message_print (0, "|
      call message_print (0, "|                                      W
      call message_print (0, "|                                     sW
      call message_print (0, "|                                     WW
      call message_print (0, "|                                    sWW
      call message_print (0, "|                                    WWW
      call message_print (0, "|                                   wWWW
      call message_print (0, "|                                   wWWWW
      call message_print (0, "|                                   WW WW
```

53

```
          call message_print (0, "|                                                 WW   WW
          call message_print (0, "|                                                wWW   WW
          call message_print (0, "|                                               wWW    WW
          call message_print (0, "|                                               WW     WW
          call message_print (0, "|                                               WW     WW
          call message_print (0, "|                                              WW      WW
          call message_print (0, "|                                              WW      WW
          call message_print (0, "|                                             WW       WW
          call message_print (0, "|                                             WW       WW
          call message_print (0, "|                  wwwwww                     WW       WW
          call message_print (0, "|                 WWWWWww                      WW      WW
          call message_print (0, "|                WWWWWwwwww    WW              WW
          call message_print (0, "|                wWWWwwwwwWW                   WW
          call message_print (0, "|               wWWWWWWWWWWWwWWW               WW
          call message_print (0, "|               wWWWWW         wW        WWWWWWW
          call message_print (0, "|               WWWW           wW        WW   wWWWWWWWwww
          call message_print (0, "|               WWWW                          wWWWWWWWwwww
          call message_print (0, "|              WWWW                               WWWW      WWw
          call message_print (0, "|             WWWWww                               WWWW
          call message_print (0, "|            WWWwwww                                WWWW
          call message_print (0, "|           wWWWWwww       wWWWWW
          call message_print (0, "|                     WwwwwwwwwWWW
          call message_print (0, "|
          call message_print (0, "|
          call message_print (0, "|
          call message_print (0, "|  by:    Wolfgang Kilian <kilian@hep.physik.uni-siegen.de>
          call message_print (0, "|         Thorsten Ohl    <ohl@physik.uni-wuerzburg.de>
          call message_print (0, "|         Juergen Reuter  <reuter@physik.uni-freiburg.de>
          call message_print (0, "|
          call message_print (0, "|  if you use WHIZARD please cite:
          call message_print (0, "|         W. Kilian, T. Ohl, J. Reuter,  arXiv: 0708.4233 [hep-ph]
          call message_print (0, "|         M. Moretti, T. Ohl, J. Reuter, arXiv: hep-ph/0102195
          call message_print (0, "|
          call message_print (0, "|=================================================================
          call message_print (0, "|                                WHIZARD " // WHIZARD_VERSION, unit=uni
          call message_print (0, "|=================================================================
        end subroutine msg_banner
```

There are several reasons for terminating the process at the next possible time:

⟨*Diagnostics: parameters*⟩+≡
```
    integer, parameter :: &
        & CONTINUE=0, KILLED=1, INTERRUPTED=2, &
        & OS_TIME_EXCEEDED=3, TIME_EXCEEDED=4, &
        & MAX_FILE_COUNT_EXCEEDED=5
```

⟨*Diagnostics: variables*⟩+≡
```
    integer, save :: terminate_status = CONTINUE
```

Prepare for termination:

⟨*Diagnostics: public*⟩+≡
```
    public :: terminate_soon
```

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine terminate_soon (reason)
      integer, intent(in) :: reason
```

```
          terminate_status = reason
       end subroutine terminate_soon
```

Terminate now if requested. If `n_events` is specified, we are in the simulation
pass, and we should display the number of events generated so far.

⟨*Diagnostics: public*⟩+≡
```
    public :: terminate_now_maybe
```

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine terminate_now_maybe (n_events)
      integer(i64), intent(in), optional :: n_events
      if (terminate_status /= CONTINUE) then
         if (present (n_events)) then
            write (msg_buffer, "(1x,A,1x,I12,1x,A)")  &
                 & "Generated", n_events, "events."
            call msg_message
         end if
         select case (terminate_status)
         case (KILLED)
            call msg_terminate (" *** Process killed by external signal: Terminating.")
         case (INTERRUPTED)
            call msg_terminate (" *** Process interrupted by external signal: Terminating.")
         case (OS_TIME_EXCEEDED)
            call msg_terminate (" *** Process received signal SIGXCPU (time exceeded):  Terminating.
         case (TIME_EXCEEDED)
            call msg_terminate (" *** User-defined time limit exceeded:  Terminating.")
         case (MAX_FILE_COUNT_EXCEEDED)
            call msg_terminate (" *** User-defined limit for event file count exceeded:  Terminating
         case default
            call msg_bug (" Process terminates for unknown reason")
         end select
      end if
    end subroutine terminate_now_maybe
```

2.7.1   Logfile

All screen output should be duplicated in the logfile, unless requested otherwise.

⟨*Diagnostics: variables*⟩+≡
```
    integer, save :: log_unit = -1
```

⟨*Diagnostics: public*⟩+≡
```
    public :: logfile_init
```

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine logfile_init (filename)
      type(string_t), intent(in) :: filename
      call msg_message ("Writing log to '" // char (filename) // "'")
      log_unit = free_unit ()
      open (file = char (filename), unit = log_unit, &
            action = "write", status = "replace")
    end subroutine logfile_init
```

⟨*Diagnostics: public*⟩+≡
```
    public :: logfile_final
```

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine logfile_final ()
      if (log_unit >= 0) then
         close (log_unit)
         log_unit = -1
      end if
    end subroutine logfile_final
```

This returns the valid logfile unit only if the default is write to screen, and if `logfile` is not set false.

⟨*Diagnostics: public*⟩+≡
```
    public :: logfile_unit
```

⟨*Diagnostics: procedures*⟩+≡
```
    function logfile_unit (unit, logfile)
      integer :: logfile_unit
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: logfile
      if (present (unit)) then
         if (unit == 6) then
            logfile_unit = log_unit
         else
            logfile_unit = -1
         end if
      else if (present (logfile)) then
         if (logfile) then
            logfile_unit = log_unit
         else
            logfile_unit = -1
         end if
      else
         logfile_unit = log_unit
      end if
    end function logfile_unit
```

2.7.2 Checking values

The `expect` function does not just check a value for correctness (actually, it checks if a logical expression is true); it records its result here. If failures are present when the program terminates, the exit code is nonzero.

⟨*Diagnostics: variables*⟩+≡
```
    integer, save :: expect_total = 0
    integer, save :: expect_failures = 0
```

⟨*Diagnostics: public*⟩+≡
```
    public :: expect_record
```

⟨*Diagnostics: procedures*⟩+≡
```fortran
    subroutine expect_record (success)
      logical, intent(in) :: success
      expect_total = expect_total + 1
      if (.not. success)  expect_failures = expect_failures + 1
    end subroutine expect_record
```

⟨*Diagnostics: public*⟩+≡
```fortran
    public :: expect_clear
```
⟨*Diagnostics: procedures*⟩+≡
```fortran
    subroutine expect_clear ()
      expect_total = 0
      expect_failures = 0
    end subroutine expect_clear
```

⟨*Diagnostics: public*⟩+≡
```fortran
    public :: expect_summary
```
⟨*Diagnostics: procedures*⟩+≡
```fortran
    subroutine expect_summary (unit)
      integer, intent(in), optional :: unit
      if (expect_total /= 0) then
         call msg_message ("Summary of value checks:", unit)
         write (msg_buffer, "(2x,A,1x,I0,1x,A,1x,A,1x,I0)") &
              "Failures:", expect_failures, "/", "Total:", expect_total
         call msg_message (unit=unit)
      end if
    end subroutine expect_summary
```

2.8   C wrapper for printf

The `printf` family of functions is implemented in C with an undefined number of arguments. This is not supported by the `bind(C)` interface. We therefore write wrappers for the versions of `sprintf` that we will actually use.

⟨`sprintf_interface.c`⟩≡
```c
  /*
  ⟨File header⟩
  */
  #include <stdio.h>

  int sprintf_none(char* str, const char* format) {
    return sprintf(str, format);
  }

  int sprintf_int(char* str, const char* format, int val) {
    return sprintf(str, format, val);
  }

  int sprintf_double(char* str, const char* format, double val) {
    return sprintf(str, format, val);
  }
```

```
int sprintf_str(char* str, const char* format, const char* val) {
  return sprintf(str, format, val);
}
```

⟨*sprintf interfaces*⟩≡
```
    interface
      function sprintf_none (str, fmt) result (stat) bind(C)
        use iso_c_binding !NODEP!
        integer(c_int) :: stat
        character(c_char), dimension(*), intent(inout) :: str
        character(c_char), dimension(*), intent(in) :: fmt
      end function sprintf_none
    end interface

    interface
      function sprintf_int (str, fmt, val) result (stat) bind(C)
        use iso_c_binding !NODEP!
        integer(c_int) :: stat
        character(c_char), dimension(*), intent(inout) :: str
        character(c_char), dimension(*), intent(in) :: fmt
        integer(c_int), value :: val
      end function sprintf_int
    end interface

    interface
      function sprintf_double (str, fmt, val) result (stat) bind(C)
        use iso_c_binding !NODEP!
        integer(c_int) :: stat
        character(c_char), dimension(*), intent(inout) :: str
        character(c_char), dimension(*), intent(in) :: fmt
        real(c_double), value :: val
      end function sprintf_double
    end interface

    interface
      function sprintf_str(str, fmt, val) result (stat) bind(C)
        use iso_c_binding !NODEP!
        integer(c_int) :: stat
        character(c_char), dimension(*), intent(inout) :: str
        character(c_char), dimension(*), intent(in) :: fmt
        character(c_char), dimension(*), intent(in) :: val
      end function sprintf_str
    end interface
```

2.9   Interface for formatted I/O

For access to formatted printing (possibly input), we interface the C `printf`
family of functions. There are two important issues here:

1. `printf` takes an arbitrary number of arguments, relying on the C stack.
   This is not interoperable. We interface it with C wrappers that output a

single integer, real or string and restrict the allowed formats accordingly.

2. Restricting format strings is essential also for preventing format string attacks. Allowing arbitrary format string would create a real security hole in a Fortran program.

3. The string returned by `sprintf` must be allocated to the right size.

⟨**formats.f90**⟩≡
　⟨*File header*⟩

　`module formats`

　　`use iso_c_binding !NODEP!`
　⟨*Use kinds*⟩
　⟨*Use strings*⟩
　⟨*Use file utils*⟩
　　`use diagnostics !NODEP!`

　⟨*Standard module head*⟩

　⟨*Formats: public*⟩

　⟨*Formats: parameters*⟩

　⟨*Formats: types*⟩

　⟨*Formats: interfaces*⟩

　`contains`

　⟨*Formats: procedures*⟩

　`end module formats`

2.9.1　Parsing a C format string

The C format string contains characters and format conversion specifications. The latter are initiated by a `%` sign. If the next letter is also a `%`, a percent sign is printed and no conversion is done. Otherwise, a conversion is done and applied to the next argument in the argument list. First comes an optional flag (`#`, `0`, `-`, `+`, or space), an optional field width (decimal digits starting not with zero), an optional precision (period, then another decimal digit string), a length modifier (irrelevant for us, therefore not supported), and a conversion specifier: `d` or `i` for integer; `e`, `f`, `g` (also upper case) for double-precision real, `s` for a string.

　We explicitly exclude all other conversion specifiers, and we check the specifiers against the actual arguments.

A type for passing arguments

This is a polymorphic type that can hold integer, real (double), and string arguments.
⟨*Formats: parameters*⟩≡

```
      integer, parameter :: ARGTYPE_NONE = 0
      integer, parameter :: ARGTYPE_LOG = 1
      integer, parameter :: ARGTYPE_INT = 2
      integer, parameter :: ARGTYPE_REAL = 3
      integer, parameter :: ARGTYPE_STR = 4
```

The integer and real entries are actually scalars, but we avoid relying on the allocatable-scalar feature and make them one-entry arrays. The character entry is a real array which is a copy of the string.

Logical values are mapped to strings (true or false), so this type parameter value is mostly unused.

⟨*Formats: public*⟩≡
```
      public :: sprintf_arg_t
```

⟨*Formats: types*⟩≡
```
      type :: sprintf_arg_t
        private
        integer :: type = ARGTYPE_NONE
        integer(c_int), dimension(:), allocatable :: ival
        real(c_double), dimension(:), allocatable :: rval
        character(c_char), dimension(:), allocatable :: sval
      end type sprintf_arg_t
```

⟨*Formats: public*⟩+≡
```
      public :: sprintf_arg_init
```

⟨*Formats: interfaces*⟩≡
```
      interface sprintf_arg_init
         module procedure sprintf_arg_init_log
         module procedure sprintf_arg_init_int
         module procedure sprintf_arg_init_real
         module procedure sprintf_arg_init_str
      end interface
```

⟨*Formats: procedures*⟩≡
```
      subroutine sprintf_arg_init_log (arg, lval)
        type(sprintf_arg_t), intent(out) :: arg
        logical, intent(in) :: lval
        arg%type = ARGTYPE_STR
        if (lval) then
           allocate (arg%sval (5))
           arg%sval = (/ 't', 'r', 'u', 'e', c_null_char /)
        else
           allocate (arg%sval (6))
           arg%sval = (/ 'f', 'a', 'l', 's', 'e', c_null_char /)
        end if
      end subroutine sprintf_arg_init_log

      subroutine sprintf_arg_init_int (arg, ival)
        type(sprintf_arg_t), intent(out) :: arg
        integer, intent(in) :: ival
        arg%type = ARGTYPE_INT
        allocate (arg%ival (1))
        arg%ival = ival
```

```
      end subroutine sprintf_arg_init_int

    subroutine sprintf_arg_init_real (arg, rval)
      type(sprintf_arg_t), intent(out) :: arg
      real(default), intent(in) :: rval
      arg%type = ARGTYPE_REAL
      allocate (arg%rval (1))
      arg%rval = rval
    end subroutine sprintf_arg_init_real

    subroutine sprintf_arg_init_str (arg, sval)
      type(sprintf_arg_t), intent(out) :: arg
      type(string_t), intent(in) :: sval
      integer :: i
      arg%type = ARGTYPE_STR
      allocate (arg%sval (len (sval) + 1))
      do i = 1, len (sval)
          arg%sval(i) = extract (sval, i, i)
      end do
      arg%sval(len (sval) + 1) = c_null_char
    end subroutine sprintf_arg_init_str
```

⟨*Formats: procedures*⟩+≡
```
    subroutine sprintf_arg_write (arg, unit)
      type(sprintf_arg_t), intent(in) :: arg
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      select case (arg%type)
      case (ARGTYPE_NONE)
        write (u, *) "[none]"
      case (ARGTYPE_INT)
        write (u, "(1x,A,1x)", advance = "no")  "[int]"
        write (u, *)  arg%ival
      case (ARGTYPE_REAL)
        write (u, "(1x,A,1x)", advance = "no")  "[real]"
        write (u, *)  arg%rval
      case (ARGTYPE_STR)
        write (u, "(1x,A,1x,A)", advance = "no")  "[string]", '"'
        write (u, *)  arg%rval, '"'
      end select
    end subroutine sprintf_arg_write
```

Return an upper bound for the length of the printed version; in case of strings
the result is exact.

⟨*Formats: procedures*⟩+≡
```
    elemental function sprintf_arg_get_length (arg) result (length)
      integer :: length
      type(sprintf_arg_t), intent(in) :: arg
      select case (arg%type)
      case (ARGTYPE_INT)
          length = log10 (real (huge (arg%ival(1)))) + 2
      case (ARGTYPE_REAL)
```

```
         length = log10 (real (radix (arg%rval(1))) ** digits (arg%rval(1))) + 8
      case (ARGTYPE_STR)
         length = size (arg%sval)
      case default
         length = 0
      end select
   end function sprintf_arg_get_length
```

⟨*Formats: procedures*⟩+≡

```
   subroutine sprintf_arg_apply_sprintf (arg, fmt, result, actual_length)
     type(sprintf_arg_t), intent(in) :: arg
     character(c_char), dimension(:), intent(in) :: fmt
     character(c_char), dimension(:), intent(inout) :: result
     integer, intent(out) :: actual_length
     integer(c_int) :: ival
     real(c_double) :: rval
     select case (arg%type)
     case (ARGTYPE_NONE)
       actual_length = sprintf_none (result, fmt)
     case (ARGTYPE_INT)
       ival = arg%ival(1)
       actual_length = sprintf_int (result, fmt, ival)
     case (ARGTYPE_REAL)
       rval = arg%rval(1)
       actual_length = sprintf_double (result, fmt, rval)
     case (ARGTYPE_STR)
       actual_length = sprintf_str (result, fmt, arg%sval)
     case default
       call msg_bug ("sprintf_arg_apply_sprintf called with illegal type")
     end select
     if (actual_length < 0) then
        write (msg_buffer, *) "Format: '", fmt, "'"
        call msg_message ()
        write (msg_buffer, *) "Output: '", result, "'"
        call msg_message ()
        call msg_error ("I/O error in sprintf call")
        actual_length = 0
     end if
   end subroutine sprintf_arg_apply_sprintf
```

Container type for the output

There is a procedure which chops the format string into pieces that contain at
most one conversion specifier. Pairing this with a `sprintf_arg` object, we get
the actual input to the `sprintf` interface. The type below holds this input and
can allocate the output string.

⟨*Formats: types*⟩+≡

```
   type :: sprintf_interface_t
     private
     character(c_char), dimension(:), allocatable :: input_fmt
     type(sprintf_arg_t) :: arg
     character(c_char), dimension(:), allocatable :: output_str
```

```
      integer :: output_str_len = 0
    end type sprintf_interface_t
```

⟨*Formats: procedures*⟩+≡
```
    subroutine sprintf_interface_init (intf, fmt, arg)
      type(sprintf_interface_t), intent(out) :: intf
      type(string_t), intent(in) :: fmt
      type(sprintf_arg_t), intent(in) :: arg
      integer :: fmt_len, i
      fmt_len = len (fmt)
      allocate (intf%input_fmt (fmt_len + 1))
      do i = 1, fmt_len
         intf%input_fmt(i) = extract (fmt, i, i)
      end do
      intf%input_fmt(fmt_len+1) = c_null_char
      intf%arg = arg
      allocate (intf%output_str (len (fmt) + sprintf_arg_get_length (arg) + 1))
    end subroutine sprintf_interface_init
```

⟨*Formats: procedures*⟩+≡
```
    subroutine sprintf_interface_write (intf, unit)
      type(sprintf_interface_t), intent(in) :: intf
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, *) "Format string = ", '"', intf%input_fmt, '"'
      write (u, "(1x,A,1x)", advance = "no") "Argument = "
      call sprintf_arg_write (intf%arg, unit)
      if (intf%output_str_len > 0) then
         write (u, *) "Result string = ", &
              '"', intf%output_str (1:intf%output_str_len), '"'
      end if
    end subroutine sprintf_interface_write
```

Return the output string:

⟨*Formats: procedures*⟩+≡
```
    function sprintf_interface_get_result (intf) result (string)
      type(string_t) :: string
      type(sprintf_interface_t), intent(in) :: intf
      character(kind = c_char, len = max (intf%output_str_len, 0)) :: buffer
      integer :: i
      if (intf%output_str_len > 0) then
         do i = 1, intf%output_str_len
            buffer(i:i) = intf%output_str(i)
         end do
         string = buffer(1:intf%output_str_len)
      else
         string = ""
      end if
    end function sprintf_interface_get_result
```

⟨*Formats: procedures*⟩+≡

```
      subroutine sprintf_interface_apply_sprintf (intf)
        type(sprintf_interface_t), intent(inout) :: intf
        call sprintf_arg_apply_sprintf &
             (intf%arg, intf%input_fmt, intf%output_str, intf%output_str_len)
      end subroutine sprintf_interface_apply_sprintf
```

Import the interfaces defined in the previous section:

⟨*Formats: interfaces*⟩+≡
  ⟨*sprintf interfaces*⟩

Scan the format string

Chop it into pieces that contain one conversion specifier each. The zero-th piece contains the part before the first specifier. Check the specifiers and allow only the subset that we support. Also check for an exact match between conversion specifiers and input arguments. The result is an allocated array of `sprintf_interface` object; each one contains a piece of the format string and the corresponding argument.

⟨*Formats: procedures*⟩+≡
```
    subroutine chop_and_check_format_string (fmt, arg, intf)
      type(string_t), intent(in) :: fmt
      type(sprintf_arg_t), dimension(:), intent(in) :: arg
      type(sprintf_interface_t), dimension(:), intent(out), allocatable :: intf
      integer :: n_args, i
      type(string_t), dimension(:), allocatable :: split_fmt
      type(string_t) :: word, buffer, separator
      integer :: pos, length, l
      logical :: ok
      type(sprintf_arg_t) :: arg_null
      ok = .true.
      length = 0
      n_args = size (arg)
      allocate (split_fmt (0:n_args))
      split_fmt = ""
      buffer = fmt
      SCAN_ARGS: do i = 1, n_args
         FIND_CONVERSION: do
            call split (buffer, word, "%", separator=separator)
            if (separator == "") then
               call msg_message ('"' // char (fmt) // '"')
               call msg_error ("C-formatting string: " &
                    // "too few conversion specifiers in format string")
               ok = .false.;  exit SCAN_ARGS
            end if
            split_fmt(i-1) = split_fmt(i-1) // word
            if (extract (buffer, 1, 1) /= "%") then
               split_fmt(i) = "%"
               exit FIND_CONVERSION
            else
               split_fmt(i-1) = split_fmt(i-1) // "%"
            end if
```

64

```
      end do FIND_CONVERSION
   pos = verify (buffer, "#0-+ ")    ! Flag characters (zero or more)
   split_fmt(i) = split_fmt(i) // extract (buffer, 1, pos-1)
   buffer = remove (buffer, 1, pos-1)
   pos = verify (buffer, "123456890")  ! Field width
   word = extract (buffer, 1, pos-1)
   if (len (word) /= 0) then
     call read_int_from_string (word, len (word), l)
     length = length + l
   end if
   split_fmt(i) = split_fmt(i) // word
   buffer = remove (buffer, 1, pos-1)
   if (extract (buffer, 1, 1) == ".") then
      buffer = remove (buffer, 1, 1)
      pos = verify (buffer, "1234567890")   ! Precision
      split_fmt(i) = split_fmt(i) // "." // extract (buffer, 1, pos-1)
      buffer = remove (buffer, 1, pos-1)
   end if
   ! Length modifier would come here, but is not allowed
   select case (char (extract (buffer, 1, 1)))  ! conversion specifier
   case ("d", "i")
      if (arg(i)%type /= ARGTYPE_INT) then
         call msg_message ('"' // char (fmt) // '"')
         call msg_error ("C-formatting string: " &
             // "argument type mismatch: integer value expected")
         ok = .false.;  exit SCAN_ARGS
      end if
   case ("e", "E", "f", "F", "g", "G")
      if (arg(i)%type /= ARGTYPE_REAL) then
         call msg_message ('"' // char (fmt) // '"')
         call msg_error ("C-formatting string: " &
             // "argument type mismatch: real value expected")
         ok = .false.;  exit SCAN_ARGS
      end if
   case ("s")
      if (arg(i)%type /= ARGTYPE_STR) then
         call msg_message ('"' // char (fmt) // '"')
         call msg_error ("C-formatting string: " &
             // "argument type mismatch: logical or string value expected")
         ok = .false.;  exit SCAN_ARGS
      end if
   case default
      call msg_message ('"' // char (fmt) // '"')
      call msg_error ("C-formatting string: " &
          // "illegal or incomprehensible conversion specifier")
      ok = .false.;  exit SCAN_ARGS
   end select
   split_fmt(i) = split_fmt(i) // extract (buffer, 1, 1)
   buffer = remove (buffer, 1, 1)
end do SCAN_ARGS
if (ok) then
   FIND_EXTRA_CONVERSION: do
      call split (buffer, word, "%", separator=separator)
      split_fmt(n_args) = split_fmt(n_args) // word // separator
```

```
            if (separator == "")  exit FIND_EXTRA_CONVERSION
            if (extract (buffer, 1, 1) == "%") then
               split_fmt(n_args) = split_fmt(n_args) // "%"
               buffer = remove (buffer, 1, 1)
            else
               call msg_message ('"' // char (fmt) // '"')
               call msg_error ("C-formatting string: " &
                    // "too many conversion specifiers in format string")
               ok = .false.;  exit FIND_EXTRA_CONVERSION
            end if
         end do FIND_EXTRA_CONVERSION
         split_fmt(n_args) = split_fmt(n_args) // buffer
         allocate (intf (0:n_args))
         call sprintf_interface_init (intf(0), split_fmt(0), arg_null)
         do i = 1, n_args
            call sprintf_interface_init (intf(i), split_fmt(i), arg(i))
         end do
      else
         allocate (intf (0))
      end if
    contains
      subroutine read_int_from_string (word, length, l)
        type(string_t), intent(in) :: word
        integer, intent(in) :: length
        integer, intent(out) :: l
        character(len=length) :: buffer
        buffer = word
        read (buffer, *) l
      end subroutine read_int_from_string
    end subroutine chop_and_check_format_string
```

2.9.2 API

⟨Formats: public⟩+≡
```
    public :: sprintf
```

⟨Formats: procedures⟩+≡
```
    function sprintf (fmt, arg) result (string)
      type(string_t) :: string
      type(string_t), intent(in) :: fmt
      type(sprintf_arg_t), dimension(:), intent(in) :: arg
      type(sprintf_interface_t), dimension(:), allocatable :: intf
      integer :: i
      string = ""
      call chop_and_check_format_string (fmt, arg, intf)
      if (size (intf) > 0) then
         do i = 0, ubound (intf, 1)
            call sprintf_interface_apply_sprintf (intf(i))
            string = string // sprintf_interface_get_result (intf(i))
         end do
      end if
    end function sprintf
```

2.9.3 Test

⟨*Formats: public*⟩+≡

```
    public :: format_test
```

⟨*Formats: procedures*⟩+≡

```
    subroutine format_test ()
      use limits, only: EOR !NODEP!
      type(string_t) :: fmt
      type(sprintf_arg_t), dimension(:), allocatable :: arg
      integer :: n_args, i, type
      logical :: lval
      integer :: ival
      real(default) :: rval
      integer :: iostat
      character(80) :: buffer
      type(string_t) :: string
      do
         print *, "Format string:"
         call get (fmt, iostat=iostat)
         select case (iostat)
         case (0, EOR)
         case default
           return
         end select
         print *, "Number of args:"
         read (*,*)  n_args
         allocate (arg (n_args))
         do i = 1, n_args
            print *, "Argument (type, value) = "
            read (*, *)  type, buffer
            select case (type)
            case (ARGTYPE_LOG)
              read (buffer, *)  lval
              call sprintf_arg_init (arg(i), lval)
            case (ARGTYPE_INT)
              read (buffer, *)  ival
              call sprintf_arg_init (arg(i), ival)
            case (ARGTYPE_REAL)
              read (buffer, *)  rval
              call sprintf_arg_init (arg(i), rval)
            case (ARGTYPE_STR)
              call sprintf_arg_init (arg(i), var_str (trim (buffer)))
            end select
         end do
         string = sprintf (fmt, arg)
         print *, "Result: '", char (string), "'"
         deallocate (arg)
      end do
    end subroutine format_test
```

2.10 Bytes and such

In a few instances we will need the notion of a byte (8-bit) and a word (32 bit), even a 64-bit word. A block of 512 bit is also needed (for MD5).

We rely on integers up to 64 bit being supported by the processor. The main difference to standard integers is the interpretation as unsigned integers.

⟨bytes.f90⟩≡
  ⟨*File header*⟩

  ```
  module bytes

    use kinds, only: i8, i32, i64 !NODEP!
  ```
  ⟨*Use file utils*⟩

  ⟨*Standard module head*⟩

  ⟨*Bytes: public*⟩

  ⟨*Bytes: types*⟩

  ⟨*Bytes: parameters*⟩

  ⟨*Bytes: interfaces*⟩

  ```
  contains
  ```

  ⟨*Bytes: procedures*⟩

  ```
  end module bytes
  ```

2.10.1 8-bit words: bytes

This is essentially a wrapper around 8-bit integers. The wrapper emphasises their special interpretation as a sequence of bits. However, we interpret bytes as unsigned integers.

⟨*Bytes: public*⟩≡
```
    public :: byte_t
```

⟨*Bytes: types*⟩≡
```
    type :: byte_t
       private
       integer(i8) :: i
    end type byte_t
```

⟨*Bytes: public*⟩+≡
```
    public :: byte_zero
```

⟨*Bytes: parameters*⟩≡
```
    type(byte_t), parameter :: byte_zero = byte_t (0_i8)
```

Set a byte from 8-bit integer:

⟨*Bytes: public*⟩+≡
```
    public :: assignment(=)
```

⟨*Bytes: interfaces*⟩≡
```
    interface assignment(=)
        module procedure set_byte_from_i8
    end interface
```

⟨*Bytes: procedures*⟩≡
```
    subroutine set_byte_from_i8 (b, i)
      type(byte_t), intent(out) :: b
      integer(i8), intent(in) :: i
      b%i = i
    end subroutine set_byte_from_i8
```

Write a byte in one of two formats: either as a hexadecimal number (two digits, default) or as a decimal number (one to three digits). The decimal version is nontrivial because bytes are unsigned integers. Optionally append a newline.

⟨*Bytes: public*⟩+≡
```
    public :: byte_write
```

⟨*Bytes: interfaces*⟩+≡
```
    interface byte_write
        module procedure byte_write_unit, byte_write_string
    end interface
```

⟨*Bytes: procedures*⟩+≡
```
    subroutine byte_write_unit (b, unit, decimal, newline)
      type(byte_t), intent(in), optional :: b
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: decimal, newline
      logical :: dc, nl
      type(word32_t) :: w
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      dc = .false.;  if (present (decimal))  dc = decimal
      nl = .false.;  if (present (newline))  nl = newline
      if (dc) then
         w = b
         write (u, '(I3)', advance='no')  w%i
      else
         write (u, 'z2.2)', advance='no')  b%i
      end if
      if (nl) write (u, *)
    end subroutine byte_write_unit
```

The string version is hex-only

⟨*Bytes: procedures*⟩+≡
```
    subroutine byte_write_string (b, s)
      type(byte_t), intent(in) :: b
      character(len=2), intent(inout) :: s
      write (s, '(z2.2)')  b%i
    end subroutine byte_write_string
```

2.10.2 32-bit words

This is not exactly a 32-bit integer. A word is to be filled with bytes, and it may be partially filled. The filling is done lowest-byte first, highest-byte last. We count the bits, so `fill` should be either 0, 8, 16, 24, or 32. In printing words, we correspondingly distinguish between printing zeros and printing blanks.

⟨*Bytes: public*⟩+≡
```
public :: word32_t
```

⟨*Bytes: types*⟩+≡
```
type :: word32_t
   private
   integer(i32) :: i
   integer :: fill = 0
end type word32_t
```

Assignment: the word is filled by inserting a 32-bit integer

⟨*Bytes: interfaces*⟩+≡
```
interface assignment(=)
   module procedure word32_set_from_i32
   module procedure word32_set_from_byte
end interface
```

⟨*Bytes: procedures*⟩+≡
```
subroutine word32_set_from_i32 (w, i)
  type(word32_t), intent(out) :: w
  integer(i32), intent(in) :: i
  w%i = i
  w%fill = 32
end subroutine word32_set_from_i32
```

Filling with a 8-bit integer is slightly tricky, because in this interpretation integers are unsigned.

⟨*Bytes: procedures*⟩+≡
```
subroutine word32_set_from_byte (w, b)
  type(word32_t), intent(out) :: w
  type(byte_t), intent(in) :: b
  if (b%i >= 0_i8) then
     w%i = b%i
  else
     w%i = 2_i32*(huge(0_i8)+1_i32) + b%i
  end if
  w%fill = 32
end subroutine word32_set_from_byte
```

Check the fill status

⟨*Bytes: public*⟩+≡
```
public :: word32_empty, word32_filled, word32_fill
```

⟨*Bytes: procedures*⟩+≡
```
function word32_empty (w)
  type(word32_t), intent(in) :: w
  logical :: word32_empty
```

```
    word32_empty = (w%fill == 0)
end function word32_empty

function word32_filled (w)
  type(word32_t), intent(in) :: w
  logical :: word32_filled
  word32_filled = (w%fill == 32)
end function word32_filled

function word32_fill (w)
  type(word32_t), intent(in) :: w
  integer :: word32_fill
  word32_fill = w%fill
end function word32_fill
```

Partial assignment: append a byte to a partially filled word. (Note: no assignment if the word is filled, so check this before if necessary.)

⟨*Bytes: public*⟩+≡
```
    public :: word32_append_byte
```

⟨*Bytes: procedures*⟩+≡
```
subroutine word32_append_byte (w, b)
  type(word32_t), intent(inout) :: w
  type(byte_t), intent(in) :: b
  type(word32_t) :: w1
  if (.not. word32_filled (w)) then
     w1 = b
     call mvbits (w1%i, 0, 8, w%i, w%fill)
     w%fill = w%fill + 8
  end if
end subroutine word32_append_byte
```

Extract a byte from a word. The argument `i` is the position, which may be 0, 1, 2, or 3.

⟨*Bytes: public*⟩+≡
```
    public :: byte_from_word32
```

⟨*Bytes: procedures*⟩+≡
```
function byte_from_word32 (w, i) result (b)
  type(word32_t), intent(in) :: w
  integer, intent(in) :: i
  type(byte_t) :: b
  integer(i32) :: j
  j = 0
  if (i >= 0 .and. i*8 < w%fill) then
     call mvbits (w%i, i*8, 8, j, 0)
  end if
  b%i = j
end function byte_from_word32
```

Write a word to file or STDOUT. We understand words as unsigned integers, therefore we cannot use the built-in routine unchanged. However, we can make use of the existence of 64-bit integers and their output routine.

In hexadecimal format, the default version prints eight hex characters, highest-first. The **bytes** version prints four bytes (two-hex characters), lowest first, with spaces in-between. The decimal bytes version is analogous. In the **bytes** version, missing bytes are printed as whitespace.

⟨*Bytes: public*⟩+≡
```
public :: word32_write
```

⟨*Bytes: interfaces*⟩+≡
```
interface word32_write
   module procedure word32_write_unit
end interface
```

⟨*Bytes: procedures*⟩+≡
```
subroutine word32_write_unit (w, unit, bytes, decimal, newline)
  type(word32_t), intent(in) :: w
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: bytes, decimal, newline
  logical :: dc, by, nl
  type(word64_t) :: ww
  integer :: i, u
  u = output_unit (unit);  if (u < 0)  return
  by = .false.;  if (present (bytes))   by = bytes
  dc = .false.;  if (present (decimal)) dc = decimal
  nl = .false.;  if (present (newline)) nl = newline
  if (by) then
     do i = 0, 3
        if (i>0)  write (u, '(1x)', advance='no')
        if (8*i < w%fill) then
           call byte_write (byte_from_word32 (w, i), unit, decimal=decimal)
        else if (dc) then
           write (u, '(3x)', advance='no')
        else
           write (u, '(2x)', advance='no')
        end if
     end do
  else if (dc) then
     ww = w
     write (u, '(I10)', advance='no') ww%i
  else
     select case (w%fill)
     case ( 0)
     case ( 8);  write (6, '(1x,z8.2)', advance='no') ibits (w%i, 0, 8)
     case (16);  write (6, '(1x,z8.4)', advance='no') ibits (w%i, 0,16)
     case (24);  write (6, '(1x,z8.6)', advance='no') ibits (w%i, 0,24)
     case (32);  write (6, '(1x,z8.8)', advance='no') ibits (w%i, 0,32)
     end select
  end if
  if (nl) write (u, *)
end subroutine word32_write_unit
```

2.10.3 Operations on 32-bit words

Define the usual logical operations, as well as addition (mod 2^{32}). We assume that all operands are completely filled.

⟨*Bytes: public*⟩+≡
```
    public :: not, ior, ieor, iand, ishftc, operator(+)
```

⟨*Bytes: interfaces*⟩+≡
```
    interface not
       module procedure word_not
    end interface
    interface ior
       module procedure word_or
    end interface
    interface ieor
       module procedure word_eor
    end interface
    interface iand
       module procedure word_and
    end interface
    interface ishftc
       module procedure word_shftc
    end interface
    interface operator(+)
       module procedure word_add
    end interface
```

⟨*Bytes: procedures*⟩+≡
```
    function word_not (w1) result (w2)
      type(word32_t), intent(in) :: w1
      type(word32_t) :: w2
      w2 = not (w1%i)
    end function word_not

    function word_or (w1, w2) result (w3)
      type(word32_t), intent(in) :: w1, w2
      type(word32_t) :: w3
      w3 = ior (w1%i, w2%i)
    end function word_or

    function word_eor (w1, w2) result (w3)
      type(word32_t), intent(in) :: w1, w2
      type(word32_t) :: w3
      w3 = ieor (w1%i, w2%i)
    end function word_eor

    function word_and (w1, w2) result (w3)
      type(word32_t), intent(in) :: w1, w2
      type(word32_t) :: w3
      w3 = iand (w1%i, w2%i)
    end function word_and

    function word_shftc (w1, s) result (w2)
      type(word32_t), intent(in) :: w1
      integer, intent(in) :: s
```

```
    type(word32_t) :: w2
    w2 = ishftc (w1%i, s, 32)
end function word_shftc


function word_add (w1, w2) result (w3)
  type(word32_t), intent(in) :: w1, w2
  type(word32_t) :: w3
  w3 = w1%i + w2%i
end function word_add
```

2.10.4 64-bit words

These objects consist of two 32-bit words. They thus can hold integer numbers larger than 2^{32} (to be exact, 2^{31} since FORTRAN integers are signed). The order is low-word, high-word.

⟨*Bytes: public*⟩+≡
```
    public :: word64_t
```

⟨*Bytes: types*⟩+≡
```
    type :: word64_t
       private
       integer(i64) :: i
    end type word64_t
```

Set a 64 bit word:

⟨*Bytes: interfaces*⟩+≡
```
    interface assignment(=)
       module procedure word64_set_from_i64
       module procedure word64_set_from_word32
    end interface
```

⟨*Bytes: procedures*⟩+≡
```
    subroutine word64_set_from_i64 (ww, i)
      type(word64_t), intent(out) :: ww
      integer(i64), intent(in) :: i
      ww%i = i
    end subroutine word64_set_from_i64
```

Filling with a 32-bit word:

⟨*Bytes: procedures*⟩+≡
```
    subroutine word64_set_from_word32 (ww, w)
      type(word64_t), intent(out) :: ww
      type(word32_t), intent(in) :: w
      if (w%i >= 0_i32) then
         ww%i = w%i
      else
         ww%i = 2_i64*(huge(0_i32)+1_i64) + w%i
      end if
    end subroutine word64_set_from_word32
```

Extract a byte from a word. The argument `i` is the position, which may be between 0 and 7.

⟨*Bytes: public*⟩+≡
```
    public :: byte_from_word64, word32_from_word64
```

⟨*Bytes: procedures*⟩+≡
```
    function byte_from_word64 (ww, i) result (b)
      type(word64_t), intent(in) :: ww
      integer, intent(in) :: i
      type(byte_t) :: b
      integer(i64) :: j
      j = 0
      if (i >= 0 .and. i*8 < 64) then
         call mvbits (ww%i, i*8, 8, j, 0)
      end if
      b%i = j
    end function byte_from_word64
```

Extract a 32-bit word from a 64-bit word. The position is either 0 or 1.

⟨*Bytes: procedures*⟩+≡
```
    function word32_from_word64 (ww, i) result (w)
      type(word64_t), intent(in) :: ww
      integer, intent(in) :: i
      type(word32_t) :: w
      integer(i64) :: j
      j = 0
      select case (i)
      case (0);  call mvbits (ww%i,  0, 32, j, 0)
      case (1);  call mvbits (ww%i, 32, 32, j, 0)
      end select
      w = int (j, kind=i32)
    end function word32_from_word64
```

Print a 64-bit word. Decimal version works up to 2^{63}. The `words` version uses the 'word32' printout, separated by two spaces. The low-word is printed first. The `bytes` version also uses the 'word32' printout. This implies that the lowest byte is first. The default version prints a hexadecimal number without spaces, highest byte first.

⟨*Bytes: public*⟩+≡
```
    public :: word64_write
```

⟨*Bytes: interfaces*⟩+≡
```
    interface word64_write
       module procedure word64_write_unit
    end interface
```

⟨*Bytes: procedures*⟩+≡
```
    subroutine word64_write_unit (ww, unit, words, bytes, decimal, newline)
      type(word64_t), intent(in) :: ww
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: words, bytes, decimal, newline
      logical :: wo, by, dc, nl
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
```

```
      wo = .false.;  if (present (words))    wo = words
      by = .false.;  if (present (bytes))    by = bytes
      dc = .false.;  if (present (decimal))  dc = decimal
      nl = .false.;  if (present (newline))  nl = newline
      if (wo .or. by) then
         call word32_write_unit (word32_from_word64 (ww, 0), unit, by, dc)
         write (u, '(2x)', advance='no')
         call word32_write_unit (word32_from_word64 (ww, 1), unit, by, dc)
      else if (dc) then
         write (u, '(I19)', advance='no') ww%i
      else
         write (u, '(Z16)', advance='no') ww%i
      end if
      if (nl) write (u, *)
   end subroutine word64_write_unit
```

2.11   MD5 Checksums

It is a bit of an overkill, but implementing MD5 checksums allows us to check
input/file integrity on the basis of a well-known standard. The building blocks
have been introduced in the `bytes` module.

⟨md5.f90⟩≡
  ⟨*File header*⟩

```
  module md5

    use kinds, only: i8, i32, i64 !NODEP!
```
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use bytes
    use limits, only: LF, EOR, EOF !NODEP!
```

  ⟨*Standard module head*⟩

  ⟨*MD5: public*⟩

  ⟨*MD5: types*⟩

  ⟨*MD5: variables*⟩

  ⟨*MD5: interfaces*⟩

```
  contains
```

  ⟨*MD5: procedures*⟩

```
  end module md5
```

2.11.1 Blocks

A block is a sequence of 16 words (64 bytes or 512 bits). We anticipate that blocks will be linked, so include a pointer to the next block. There is a fill status (word counter), as there is one for each word. The fill status is equal to the number of bytes that are in, so it may be between 0 and 64.

⟨*MD5: types*⟩≡
```
type :: block_t
   private
   type(word32_t), dimension(0:15) :: w
   type(block_t), pointer :: next => null ()
   integer :: fill = 0
end type block_t
```

Check if a block is completely filled or empty:

⟨*MD5: procedures*⟩≡
```
function block_is_empty (b)
  type(block_t), intent(in) :: b
  logical :: block_is_empty
  block_is_empty = (b%fill == 0 .and. word32_empty (b%w(0)))
end function block_is_empty

function block_is_filled (b)
  type(block_t), intent(in) :: b
  logical :: block_is_filled
  block_is_filled = (b%fill == 64)
end function block_is_filled
```

Append a single byte to a block. Works only if the block is not yet filled.

⟨*MD5: procedures*⟩+≡
```
subroutine block_append_byte (bl, by)
  type(block_t), intent(inout) :: bl
  type(byte_t), intent(in) :: by
  if (.not. block_is_filled (bl)) then
     call word32_append_byte (bl%w(bl%fill/4), by)
     bl%fill = bl%fill + 1
  end if
end subroutine block_append_byte
```

The printing routine allows for printing as sequences of words or bytes, decimal or hex.

⟨*MD5: interfaces*⟩≡
```
interface block_write
   module procedure block_write_unit
end interface
```

⟨*MD5: procedures*⟩+≡
```
subroutine block_write_unit (b, unit, bytes, decimal)
  type(block_t), intent(in) :: b
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: bytes, decimal
  logical :: by, dc
  integer :: i, u
```

```
      u = output_unit (unit);  if (u < 0)  return
      by = .false.;  if (present (bytes))    by = bytes
      dc = .false.;  if (present (decimal))  dc = decimal
      do i = 0, b%fill/4 - 1
         call newline_or_blank (u, i, by, dc)
         call word32_write (b%w(i), unit, bytes, decimal)
      end do
      if (.not. block_is_filled (b)) then
         i = b%fill/4
         if (.not. word32_empty (b%w(i))) then
            call newline_or_blank (u, i, by, dc)
            call word32_write (b%w(i), unit, bytes, decimal)
         end if
      end if
      write (u, *)
   contains
      subroutine newline_or_blank (u, i, bytes, decimal)
        integer, intent(in) :: u, i
        logical, intent(in) :: bytes, decimal
        if (decimal) then
           select case (i)
           case (0)
           case (2,4,6,8,10,12,14);  write (u, *)
           case default
              write (u, '(2x)', advance='no')
           end select
        else if (bytes) then
           select case (i)
           case (0)
           case (4,8,12);  write (u, *)
           case default
              write (u, '(2x)', advance='no')
           end select
        else
           if (i == 8)  write (u, *)
        end if
      end subroutine newline_or_blank
   end subroutine block_write_unit
```

2.11.2 Messages

A message (within this module) is a linked list of blocks.

⟨*MD5: types*⟩+≡

```
   type :: message_t
     private
     type(block_t), pointer :: first => null ()
     type(block_t), pointer :: last => null ()
     integer :: n_blocks = 0
   end type message_t
```

Clear the message list

⟨*MD5: procedures*⟩+≡

```
subroutine message_clear (m)
  type(message_t), intent(inout) :: m
  type(block_t), pointer :: b
  nullify (m%last)
  do
     b => m%first
     if (.not.(associated (b))) exit
     m%first => b%next
     deallocate (b)
  end do
  m%n_blocks = 0
end subroutine message_clear
```

Append an empty block to the message list

⟨*MD5: procedures*⟩+≡

```
subroutine message_append_new_block (m)
  type(message_t), intent(inout) :: m
  if (associated (m%last)) then
     allocate (m%last%next)
     m%last => m%last%next
     m%n_blocks = m%n_blocks + 1
  else
     allocate (m%first)
     m%last => m%first
     m%n_blocks = 1
  end if
end subroutine message_append_new_block
```

Initialize: clear and allocate the first (empty) block.

⟨*MD5: procedures*⟩+≡

```
subroutine message_init (m)
  type(message_t), intent(inout) :: m
  call message_clear (m)
  call message_append_new_block (m)
end subroutine message_init
```

Append a single byte to a message. If necessary, allocate a new block. If the
message is empty, initialize it.

⟨*MD5: procedures*⟩+≡

```
subroutine message_append_byte (m, b)
  type(message_t), intent(inout) :: m
  type(byte_t), intent(in) :: b
  if (.not. associated (m%last)) then
     call message_init (m)
  else if (block_is_filled (m%last)) then
     call message_append_new_block (m)
  end if
  call block_append_byte (m%last, b)
end subroutine message_append_byte
```

Append zero bytes until the current block is filled up to the required position.
If we are already beyond that, append a new block and fill that one.

⟨*MD5: procedures*⟩+≡

```fortran
    subroutine message_pad_zero (m, i)
      type(message_t), intent(inout) :: m
      integer, intent(in) :: i
      type(block_t), pointer :: b
      integer :: j
      if (associated (m%last)) then
         b => m%last
         if (b%fill > i) then
            do j = b%fill + 1, 64 + i
               call message_append_byte (m, byte_zero)
            end do
         else
            do j = b%fill + 1, i
               call message_append_byte (m, byte_zero)
            end do
         end if
      end if
    end subroutine message_pad_zero
```

This returns the number of bits within a message. We need a 64-bit word for the result since it may be more than 2^{31}. This is also required by the MD5 standard.

⟨*MD5: procedures*⟩+≡

```fortran
    function message_bits (m) result (length)
      type(message_t), intent(in) :: m
      type(word64_t) :: length
      type(block_t), pointer :: b
      integer(i64) :: n_blocks_filled, n_bytes_extra
      if (m%n_blocks > 0) then
         b => m%last
         if (block_is_filled (b)) then
            n_blocks_filled = m%n_blocks
            n_bytes_extra = 0
         else
            n_blocks_filled = m%n_blocks - 1
            n_bytes_extra = b%fill
         end if
         length = n_blocks_filled * 512 + n_bytes_extra * 8
      else
         length = 0_i64
      end if
    end function message_bits
```

2.11.3 Message I/O

Append the contents of a string to a message. We first cast the character string into a 8-bit integer array and the append this byte by byte.

⟨*MD5: procedures*⟩+≡

```fortran
    subroutine message_append_string (m, s)
      type(message_t), intent(inout) :: m
      character(len=*), intent(in) :: s
```

```
      integer(i64) :: i, n_bytes
      integer(i8), dimension(:), allocatable :: buffer
      integer(i8), dimension(1) :: mold
      type(byte_t) :: b
      n_bytes = size (transfer (s, mold))
      allocate (buffer (n_bytes))
      buffer = transfer (s, mold)
      do i = 1, size (buffer)
         b = buffer(i)
         call message_append_byte (m, b)
      end do
      deallocate (buffer)
   end subroutine message_append_string
```

Append the contents of a 32-bit integer to a message. We first cast the 32-bit integer into a 8-bit integer array and the append this byte by byte.

⟨*MD5: procedures*⟩+≡
```
   subroutine message_append_i32 (m, x)
     type(message_t), intent(inout) :: m
     integer(i32), intent(in) :: x
     integer(i8), dimension(4) :: buffer
     type(byte_t) :: b
     integer :: i
     buffer = transfer (x, buffer, size(buffer))
     do i = 1, size (buffer)
        b = buffer(i)
        call message_append_byte (m, b)
     end do
   end subroutine message_append_i32
```

Append one line from file to a message. Include the newline character.

⟨*MD5: procedures*⟩+≡
```
   !   subroutine message_append_from_unit (m, u, iostat)
   !     type(message_t), intent(inout) :: m
   !     integer, intent(in) :: u
   !     integer, intent(out) :: iostat
   !     character(len=BUFFER_SIZE) :: buffer
   !     read (u, *, iostat=iostat) buffer
   !     call message_append_string (m, trim (buffer))
   !     call message_append_string (m, LF)
   !   end subroutine message_append_from_unit
```

Fill a message from file. (Each line counts as a string.)

⟨*MD5: procedures*⟩+≡
```
   !   subroutine message_read_from_file (m, f)
   !     type(message_t), intent(inout) :: m
   !     character(len=*), intent(in) :: f
   !     integer :: u, iostat
   !     u = free_unit ()
   !     open (file=f, unit=u, action='read')
   !     do
   !        call message_append_from_unit (m, u, iostat=iostat)
```

```
!        if (iostat < 0) exit
!     end do
!     close (u)
!   end subroutine message_read_from_file
```

Write a message. After each block, insert an empty line.

⟨*MD5: interfaces*⟩+≡
```
interface message_write
   module procedure message_write_unit
end interface
```

⟨*MD5: procedures*⟩+≡
```
subroutine message_write_unit (m, unit, bytes, decimal)
  type(message_t), intent(in) :: m
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: bytes, decimal
  type(block_t), pointer :: b
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  b => m%first
  if (associated (b)) then
     do
        call block_write_unit (b, unit, bytes, decimal)
        b => b%next
        if (.not. associated (b))  exit
        write (u, *)
     end do
  end if
end subroutine message_write_unit
```

2.11.4   Auxiliary functions

These four functions on three words are defined in the MD5 standard:

⟨*MD5: procedures*⟩+≡
```
function ff (x, y, z)
  type(word32_t), intent(in) :: x, y, z
  type(word32_t) :: ff
  ff = ior (iand (x, y), iand (not (x), z))
end function ff

function fg (x, y, z)
  type(word32_t), intent(in) :: x, y, z
  type(word32_t) :: fg
  fg = ior (iand (x, z), iand (y, not (z)))
end function fg

function fh (x, y, z)
  type(word32_t), intent(in) :: x, y, z
  type(word32_t) :: fh
  fh = ieor (ieor (x, y), z)
end function fh
```

```
function fi (x, y, z)
  type(word32_t), intent(in) :: x, y, z
  type(word32_t) :: fi
  fi = ieor (y, ior (x, not (z)))
end function fi
```

2.11.5 Auxiliary stuff

This defines and initializes the table of transformation constants:

⟨*MD5: variables*⟩≡
```
    type(word32_t), dimension(64), save :: t
    logical, save :: table_initialized = .false.
```

⟨*MD5: procedures*⟩+≡
```
    subroutine table_init
      type(word64_t) :: ww
      integer :: i
      if (.not.table_initialized) then
         do i = 1, 64
            ww = int (4294967296d0 * abs (sin (i * 1d0)), kind=i64)
            t(i) = word32_from_word64 (ww, 0)
         end do
         table_initialized = .true.
      end if
    end subroutine table_init
```

This encodes the message digest (4 words) into a 32-character string.

⟨*MD5: procedures*⟩+≡
```
    function digest_string (aa) result (s)
      type(word32_t), dimension (0:3), intent(in) :: aa
      character(len=32) :: s
      integer :: i, j
      do i = 0, 3
         do j = 0, 3
            call byte_write (byte_from_word32 (aa(i), j), s(i*8+j*2+1:i*8+j*2+2))
         end do
      end do
    end function digest_string
```

2.11.6 MD5 algorithm

Pad the message with a byte x80 and then pad zeros up to a full block minus two words; in these words, insert the message length (before padding) as a 64-bit word, low-word first.

⟨*MD5: procedures*⟩+≡
```
    subroutine message_pad (m)
      type(message_t), intent(inout) :: m
      type(word64_t) :: length
      integer(i8), parameter :: ipad = -128  ! z'80'
      type(byte_t) :: b
```

```
    integer :: i
    length = message_bits (m)
    b = ipad
    call message_append_byte (m, b)
    call message_pad_zero (m, 56)
    do i = 0, 7
        call message_append_byte (m, byte_from_word64 (length, i))
    end do
end subroutine message_pad
```

Apply a series of transformations onto a state `a,b,c,d`, where the transform function uses each word of the message together with the predefined words. Finally, encode the state as a 32-character string.

⟨*MD5: procedures*⟩+≡

```
subroutine message_digest (m, s)
  type(message_t), intent(in) :: m
  character(len=32), intent(out) :: s
  integer(i32), parameter :: ia =  1732584193  ! z'67452301'
  integer(i32), parameter :: ib = -271733879  ! z'efcdab89'
  integer(i32), parameter :: ic = -1732584194  ! z'98badcfe'
  integer(i32), parameter :: id =   271733878  ! z'10325476'
  type(word32_t) :: a, b, c, d
  type(word32_t) :: aa, bb, cc, dd
  type(word32_t), dimension(0:15) :: x
  type(block_t), pointer :: bl
  call table_init
  a = ia;  b = ib;  c = ic;  d = id
  bl => m%first
  do
     if (.not.associated (bl)) exit
     x = bl%w
     aa = a; bb = b; cc = c; dd = d
     call transform (ff, a, b, c, d,  0,  7,  1)
     call transform (ff, d, a, b, c,  1, 12,  2)
     call transform (ff, c, d, a, b,  2, 17,  3)
     call transform (ff, b, c, d, a,  3, 22,  4)
     call transform (ff, a, b, c, d,  4,  7,  5)
     call transform (ff, d, a, b, c,  5, 12,  6)
     call transform (ff, c, d, a, b,  6, 17,  7)
     call transform (ff, b, c, d, a,  7, 22,  8)
     call transform (ff, a, b, c, d,  8,  7,  9)
     call transform (ff, d, a, b, c,  9, 12, 10)
     call transform (ff, c, d, a, b, 10, 17, 11)
     call transform (ff, b, c, d, a, 11, 22, 12)
     call transform (ff, a, b, c, d, 12,  7, 13)
     call transform (ff, d, a, b, c, 13, 12, 14)
     call transform (ff, c, d, a, b, 14, 17, 15)
     call transform (ff, b, c, d, a, 15, 22, 16)
     call transform (fg, a, b, c, d,  1,  5, 17)
     call transform (fg, d, a, b, c,  6,  9, 18)
     call transform (fg, c, d, a, b, 11, 14, 19)
     call transform (fg, b, c, d, a,  0, 20, 20)
     call transform (fg, a, b, c, d,  5,  5, 21)
```

```
      call transform (fg, d, a, b, c, 10,  9, 22)
      call transform (fg, c, d, a, b, 15, 14, 23)
      call transform (fg, b, c, d, a,  4, 20, 24)
      call transform (fg, a, b, c, d,  9,  5, 25)
      call transform (fg, d, a, b, c, 14,  9, 26)
      call transform (fg, c, d, a, b,  3, 14, 27)
      call transform (fg, b, c, d, a,  8, 20, 28)
      call transform (fg, a, b, c, d, 13,  5, 29)
      call transform (fg, d, a, b, c,  2,  9, 30)
      call transform (fg, c, d, a, b,  7, 14, 31)
      call transform (fg, b, c, d, a, 12, 20, 32)
      call transform (fh, a, b, c, d,  5,  4, 33)
      call transform (fh, d, a, b, c,  8, 11, 34)
      call transform (fh, c, d, a, b, 11, 16, 35)
      call transform (fh, b, c, d, a, 14, 23, 36)
      call transform (fh, a, b, c, d,  1,  4, 37)
      call transform (fh, d, a, b, c,  4, 11, 38)
      call transform (fh, c, d, a, b,  7, 16, 39)
      call transform (fh, b, c, d, a, 10, 23, 40)
      call transform (fh, a, b, c, d, 13,  4, 41)
      call transform (fh, d, a, b, c,  0, 11, 42)
      call transform (fh, c, d, a, b,  3, 16, 43)
      call transform (fh, b, c, d, a,  6, 23, 44)
      call transform (fh, a, b, c, d,  9,  4, 45)
      call transform (fh, d, a, b, c, 12, 11, 46)
      call transform (fh, c, d, a, b, 15, 16, 47)
      call transform (fh, b, c, d, a,  2, 23, 48)
      call transform (fi, a, b, c, d,  0,  6, 49)
      call transform (fi, d, a, b, c,  7, 10, 50)
      call transform (fi, c, d, a, b, 14, 15, 51)
      call transform (fi, b, c, d, a,  5, 21, 52)
      call transform (fi, a, b, c, d, 12,  6, 53)
      call transform (fi, d, a, b, c,  3, 10, 54)
      call transform (fi, c, d, a, b, 10, 15, 55)
      call transform (fi, b, c, d, a,  1, 21, 56)
      call transform (fi, a, b, c, d,  8,  6, 57)
      call transform (fi, d, a, b, c, 15, 10, 58)
      call transform (fi, c, d, a, b,  6, 15, 59)
      call transform (fi, b, c, d, a, 13, 21, 60)
      call transform (fi, a, b, c, d,  4,  6, 61)
      call transform (fi, d, a, b, c, 11, 10, 62)
      call transform (fi, c, d, a, b,  2, 15, 63)
      call transform (fi, b, c, d, a,  9, 21, 64)
      a = a + aa
      b = b + bb
      c = c + cc
      d = d + dd
      bl => bl%next
    end do
    s = digest_string ((/a, b, c, d/))
contains
⟨MD5: Internal subroutine transform⟩
end subroutine message_digest
```

And this is the actual transformation that depends on one of the previous functions, four words, and three integers. The implicit arguments are `x`, the word from the message to digest, and `t`, the entry in the predefined table.

⟨*MD5: Internal subroutine transform*⟩≡

```
subroutine transform (f, a, b, c, d, k, s, i)
  interface
    function f (x, y, z)
      import word32_t
      type(word32_t), intent(in) :: x, y, z
      type(word32_t) :: f
    end function f
  end interface
  type(word32_t), intent(inout) :: a
  type(word32_t), intent(in) :: b, c, d
  integer, intent(in) :: k, s, i
  a = b + ishftc (a + f(b, c, d) + x(k) + t(i), s)
end subroutine transform
```

2.11.7   User interface

⟨*MD5: public*⟩≡

```
public :: md5sum
```

⟨*MD5: interfaces*⟩+≡

```
interface md5sum
   module procedure md5sum_from_string
   module procedure md5sum_from_unit
end interface
```

This function computes the MD5 sum of the input string and returns it as a 32-character string

⟨*MD5: procedures*⟩+≡

```
function md5sum_from_string (s) result (digest)
  character(len=*), intent(in) :: s
  character(len=32) :: digest
  type(message_t) :: m
  call message_append_string (m, s)
  call message_pad (m)
  call message_digest (m, digest)
  call message_clear (m)
end function md5sum_from_string
```

This funct. reads from unit u (an unformmated sequence of integers) and computes the MD5 sum.

⟨*MD5: procedures*⟩+≡

```
function md5sum_from_unit (u) result (digest)
  integer, intent(in) :: u
  character(len=32) :: digest
  type(message_t) :: m
  character :: char
  integer :: iostat
  READ_CHARS: do
```

```
      read (u, "(A)", advance="no", iostat=iostat)  char
      select case (iostat)
      case (0)
         call message_append_string (m, char)
      case (EOR)
         call message_append_string (m, LF)
      case (EOF)
         exit READ_CHARS
      case default
         call msg_fatal &
              ("Computing MD5 sum: I/O error while reading from scratch file")
      end select
   end do READ_CHARS
   call message_pad (m)
   call message_digest (m, digest)
   call message_clear (m)
end function md5sum_from_unit
```

This funct checks the implementation by computing the checksum of certain strings and comparing them with the known values. If some inconsistency is detected, print a warning.

⟨*MD5: public*⟩+≡
```
    public :: md5_test
```

⟨*MD5: procedures*⟩+≡
```
    subroutine md5_test
      character(32) :: s
      integer, parameter :: n = 7
      integer :: i
      character(80), dimension(n) :: teststring
      data teststring(1) /""/
      data teststring(2) /"a"/
      data teststring(3) /"abc"/
      data teststring(4) /"message digest"/
      data teststring(5) /"abcdefghijklmnopqrstuvwxyz"/
      data teststring(6) /"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789"/
      data teststring(7) /"12345678901234567890123456789012345678901234567890123456789012345678901
      character(32), dimension(n) :: result
      data result(1) /"D41D8CD98F00B204E9800998ECF8427E"/
      data result(2) /"0CC175B9C0F1B6A831C399E269772661"/
      data result(3) /"900150983CD24FB0D6963F7D28E17F72"/
      data result(4) /"F96B697D7CB7938D525A2F31AAF161D0"/
      data result(5) /"C3FCD3D76192E4007DFB496CCA67E13B"/
      data result(6) /"D174AB98D277D9F5A5611C2C9F419D9F"/
      data result(7) /"57EDF4A22BE3C955AC49DA2E2107B67A"/
      do i = 1, n
         call msg_message ("string = " // '"'// trim (teststring(i)) // '"')
         s = md5sum (trim (teststring(i)))
         call msg_message ("md5sum = " // trim (s))
         if (s == result(i)) then
            call msg_message ("-> ok")
         else
            call msg_message ("expect = " // trim (result(i)))
            call msg_bug (" MD5 sum self-test failed")
```
```

```
          end if
        end do
        call msg_message ("MD5 sum self-test successful.")
      end subroutine md5_test
```

2.12   Permutations

⟨permutations.f90⟩≡
  ⟨*File header*⟩

  ```
  module permutations
```

    ```
    use kinds, only: TC !NODEP!
```

  ⟨*Standard module head*⟩

  ⟨*Permutations: public*⟩

  ⟨*Permutations: types*⟩

  ⟨*Permutations: interfaces*⟩

  ```
  contains
```

  ⟨*Permutations: procedures*⟩

  ```
  end module permutations
```

2.12.1   Permutations

A permutation is an array of integers. Each integer between one and `size` should occur exactly once.

⟨*Permutations: public*⟩≡
```
    public :: permutation_t
```

⟨*Permutations: types*⟩≡
```
    type :: permutation_t
      private
      integer, dimension(:), allocatable :: p
    end type permutation_t
```

Initialize with the identity permutation.

⟨*Permutations: public*⟩+≡
```
    public :: permutation_init
    public :: permutation_final
```

⟨*Permutations: procedures*⟩≡
```
    elemental subroutine permutation_init (p, size)
      type(permutation_t), intent(inout) :: p
      integer, intent(in) :: size
      integer :: i
```

```
      allocate (p%p (size))
      forall (i = 1:size)
         p%p(i) = i
      end forall
    end subroutine permutation_init

    elemental subroutine permutation_final (p)
      type(permutation_t), intent(inout) :: p
      deallocate (p%p)
    end subroutine permutation_final
```

I/O:

⟨*Permutations: public*⟩+≡
```
    public :: permutation_write
```

⟨*Permutations: procedures*⟩+≡
```
    subroutine permutation_write (p, u)
      type(permutation_t), intent (in) :: p
      integer, intent(in) :: u
      integer :: i
      do i = 1, size (p%p)
         if (size (p%p) < 10) then
            write (u,"(1x,I1)", advance="no") p%p(i)
         else
            write (u,"(1x,I3)", advance="no") p%p(i)
         end if
      end do
      write (u, *)
    end subroutine permutation_write
```

Administration:

⟨*Permutations: public*⟩+≡
```
    public :: permutation_size
```

⟨*Permutations: procedures*⟩+≡
```
    elemental function permutation_size (perm) result (s)
      type(permutation_t), intent(in) :: perm
      integer :: s
      s = size (perm%p)
    end function permutation_size
```

Extract an entry in a permutation.

⟨*Permutations: public*⟩+≡
```
    public :: permute
```

⟨*Permutations: procedures*⟩+≡
```
    elemental function permute (i, p) result (j)
      integer, intent(in) :: i
      type(permutation_t), intent(in) :: p
      integer :: j
      if (i > 0 .and. i <= size (p%p)) then
         j = p%p(i)
      else
         j = 0
```

```
      end if
    end function permute
```

Check whether a permutation is valid: Each integer in the range occurs exactly once.

⟨*Permutations: public*⟩+≡
```
    public :: permutation_ok
```

⟨*Permutations: procedures*⟩+≡
```
    elemental function permutation_ok (perm) result (ok)
      type(permutation_t), intent(in) :: perm
      logical :: ok
      integer :: i
      logical, dimension(:), allocatable :: set
      ok = .true.
      allocate (set (size (perm%p)))
      set = .false.
      do i = 1, size (perm%p)
         ok = (perm%p(i) > 0 .and. perm%p(i) <= size (perm%p))
         if (.not.ok) return
         set(perm%p(i)) = .true.
      end do
      ok = all (set)
    end function permutation_ok
```

Find the permutation that transforms the second array into the first one. We assume that this is possible and unique and all bounds are set correctly.

This cannot be elemental.

⟨*Permutations: public*⟩+≡
```
    public :: permutation_find
```

⟨*Permutations: procedures*⟩+≡
```
    subroutine permutation_find (perm, a1, a2)
      type(permutation_t), intent(inout) :: perm
      integer, dimension(:), intent(in) :: a1, a2
      integer :: i, j
      if (allocated (perm%p))  deallocate (perm%p)
      allocate (perm%p (size (a1)))
      do i = 1, size (a1)
         do j = 1, size (a2)
            if (a1(i) == a2(j)) then
               perm%p(i) = j
               exit
            end if
            perm%p(i) = 0
         end do
      end do
    end subroutine permutation_find
```

Find all permutations that transform an array of integers into itself. The resulting permutation list is allocated with the correct length and filled.

The first step is to count the number of different entries in `code`. Next, we scan `code` again and assign a mask to each different entry, true for all identical entries. Finally, we recursively permute the identity for each possible mask.

The permutation is done as follows: A list of all permutations of the initial one with respect to the current mask is generated, then the permutations are generated in turn for each permutation in this list with the next mask. The result is always stored back into the main list, starting from the end of the current list.

⟨*Permutations: public*⟩+≡

```
   public :: permutation_array_make
```

⟨*Permutations: procedures*⟩+≡

```
   subroutine permutation_array_make (pa, code)
     type(permutation_t), dimension(:), allocatable, intent(out) :: pa
     integer, dimension(:), intent(in) :: code
     logical, dimension(size(code)) :: mask
     logical, dimension(:,:), allocatable :: imask
     integer, dimension(:), allocatable :: n_i
     type(permutation_t) :: p_init
     type(permutation_t), dimension(:), allocatable :: p_tmp
     integer :: psize, i, j, k, n_different, n, nn_k
     psize = size (code)
     mask = .true.
     n_different = 0
     do i=1, psize
        if (mask(i)) then
           n_different = n_different + 1
           mask = mask .and. (code /= code(i))
        end if
     end do
     allocate (imask(psize, n_different), n_i(n_different))
     mask = .true.
     k = 0
     do i=1, psize
        if (mask(i)) then
           k = k + 1
           imask(:,k) = (code == code(i))
           n_i(k) = factorial (count(imask(:,k)))
           mask = mask .and. (code /= code(i))
        end if
     end do
     n = product (n_i)
     allocate (pa (n))
     call permutation_init (p_init, psize)
     pa(1) = p_init
     nn_k = 1
     do k = 1, n_different
        allocate (p_tmp (n_i(k)))
        do i = nn_k, 1, -1
           call permutation_array_with_mask (p_tmp, imask(:,k), pa(i))
           do j = n_i(k), 1, -1
              pa((i-1)*n_i(k) + j) = p_tmp(j)
           end do
        end do
```

```
      deallocate (p_tmp)
      nn_k = nn_k * n_i(k)
    end do
    call permutation_final (p_init)
    deallocate (imask, n_i)
  end subroutine permutation_array_make
```

Make a list of permutations of the elements marked true in the `mask` array. The final permutation list must be allocated with the correct length ($n!$). The third argument is the initial permutation to start with, which must have the same length as the `mask` array (this is not checked).

⟨*Permutations: procedures*⟩+≡
```
    subroutine permutation_array_with_mask (pa, mask, p_init)
      type(permutation_t), dimension(:), intent(inout) :: pa
      logical, dimension(:), intent(in) :: mask
      type(permutation_t), intent(in) :: p_init
      integer :: plen
      integer :: i, ii, j, fac_i, k, x
      integer, dimension(:), allocatable :: index
      plen = size (pa)
      allocate (index(count(mask)))
      ii = 0
      do i = 1, size (mask)
         if (mask(i)) then
            ii = ii + 1
            index(ii) = i
         end if
      end do
      pa = p_init
      ii = 0
      fac_i = 1
      do i = 1, size (mask)
         if (mask(i)) then
            ii = ii + 1
            fac_i = fac_i * ii
            x = permute (i, p_init)
            do j = 1, plen
               k = ii - mod (((j-1)*fac_i)/plen, ii)
               call insert (pa(j), x, k, ii, index)
            end do
         end if
      end do
      deallocate (index)
    contains
      subroutine insert (p, x, k, n, index)
        type(permutation_t), intent(inout) :: p
        integer, intent(in) :: x, k, n
        integer, dimension(:), intent(in) :: index
        integer :: i
        do i = n, k+1, -1
           p%p(index(i)) = p%p(index(i-1))
        end do
        p%p(index(k)) = x
```

```
      end subroutine insert
    end subroutine permutation_array_with_mask
```

The factorial function is needed for pre-determining the number of permutations that will be generated:

⟨*Permutations: procedures*⟩+≡
```
    function factorial (n) result (f)
      integer, intent(in) :: n
      integer :: f
      integer :: i
      f = 1
      do i=2, abs(n)
         f = f*i
      end do
    end function factorial
```

2.12.2   Operations on binary codes

Binary codes are needed for phase-space trees. Since the permutation function uses permutations, and no other special type is involved, we put the functions here.

This is needed for phase space trees: permute bits in a tree binary code. If no permutation is given, leave as is. (We may want to access the permutation directly here if this is efficiency-critical.)

⟨*Permutations: public*⟩+≡
```
    public :: tc_permute
```

⟨*Permutations: procedures*⟩+≡
```
    function tc_permute (k, perm, mask_in) result (pk)
      integer(TC), intent(in) :: k, mask_in
      type(permutation_t), intent(in) :: perm
      integer(TC) :: pk
      integer :: i
      pk = iand (k, mask_in)
      do i = 1, size (perm%p)
!        if (btest(k,i-1))  pk = ibset (pk, permute (perm,i) - 1)
         if (btest(k,i-1))  pk = ibset (pk, perm%p(i)-1)
      end do
    end function tc_permute
```

This routine returns the number of set bits in the tree code value `k`. Hence, it is the number of externals connected to the current line. If `mask` is present, the complement of the tree code is also considered, and the smaller number is returned. This gives the true distance from the external states, taking into account the initial particles. The complement number is increased by one, since for a scattering diagram the vertex with the sum of all final-state codes is still one point apart from the initial particles.

⟨*Permutations: public*⟩+≡
```
    public :: tc_decay_level
```

⟨*Permutations: interfaces*⟩≡

```
    interface tc_decay_level
       module procedure decay_level_simple
       module procedure decay_level_complement
    end interface
```

⟨*Permutations: procedures*⟩+≡

```
    function decay_level_complement (k, mask) result (l)
      integer(TC), intent(in) :: k, mask
      integer :: l
      l = min (decay_level_simple (k), &
           &   decay_level_simple (ieor (k, mask)) + 1)
    end function decay_level_complement

    function decay_level_simple (k) result(l)
      integer(TC), intent(in) :: k
      integer :: l
      integer :: i
      l = 0
      do i=0, bit_size(k)-1
         if (btest(k,i)) l = l+1
      end do
    end function decay_level_simple
```

2.13   Sorting

This small module provides functions for sorting integer or real arrays.

⟨`sorting.f90`⟩≡
  ⟨*File header*⟩

  module sorting

  ⟨*Use kinds*⟩

  ⟨*Standard module head*⟩

  ⟨*Sorting: public*⟩

  ⟨*Sorting: interfaces*⟩

  contains

  ⟨*Sorting: procedures*⟩

  end module sorting

2.13.1   Implementation

The sort function returns, for a given integer or real array, the array sorted by increasing value. The current implementation is *mergesort*, which has $O(n \ln n)$ behavior in all cases, and is stable for elements of equal value.

⟨*Sorting: public*⟩≡

```
    public :: sort
```

⟨*Sorting: interfaces*⟩≡
```
    interface sort
       module procedure sort_int
       module procedure sort_real
    end interface
```

The body is identical, just the interface differs.

⟨*Sorting: procedures*⟩≡
```
    function sort_int (val_in) result (val)
      integer, dimension(:), intent(in) :: val_in
      integer, dimension(size(val_in)) :: val
    ⟨Sorting: sort⟩
    end function sort_int

    function sort_real (val_in) result (val)
      real(default), dimension(:), intent(in) :: val_in
      real(default), dimension(size(val_in)) :: val
    ⟨Sorting: sort⟩
    end function sort_real
```

⟨*Sorting: sort*⟩≡
```
    val = val_in( order (val) )
```

The `order` function returns, for a given integer or real array, the array of indices of the elements sorted by increasing value.

⟨*Sorting: public*⟩+≡
```
    public :: order
```

⟨*Sorting: interfaces*⟩+≡
```
    interface order
       module procedure order_int
       module procedure order_real
    end interface
```

⟨*Sorting: procedures*⟩+≡
```
    function order_int (val) result (idx)
      integer, dimension(:), intent(in) :: val
      integer, dimension(size(val)) :: idx
    ⟨Sorting: order⟩
    end function order_int

    function order_real (val) result (idx)
      real(default), dimension(:), intent(in) :: val
      integer, dimension(size(val)) :: idx
    ⟨Sorting: order⟩
    end function order_real
```

We start by individual elements, merge them to pairs, merge those to four-element subarrays, and so on. The last subarray can extend only up to the original array bound, of course, and the second of the subarrays to merge should contain at least one element.

⟨*Sorting: order*⟩≡

```
    integer :: n, i, s, b1, b2, e1, e2
    n = size (idx)
    forall (i = 1:n)
       idx(i) = i
    end forall
    s = 1
    do while (s < n)
       do b1 = 1, n-s, 2*s
          b2 = b1 + s
          e1 = b2 - 1
          e2 = min (e1 + s, n)
          call merge (idx(b1:e2), idx(b1:e1), idx(b2:e2), val)
       end do
       s = 2 * s
    end do
```

The merging step does the actual sorting. We take two sorted array sections and merge them to a sorted result array. We are working on the indices, and comparing is done by taking the associated `val` which is real or integer.

⟨*Sorting: interfaces*⟩+≡

```
    interface merge
       module procedure merge_int
       module procedure merge_real
    end interface
```

⟨*Sorting: procedures*⟩+≡

```
    subroutine merge_int (res, src1, src2, val)
      integer, dimension(:), intent(out) :: res
      integer, dimension(:), intent(in) :: src1, src2
      integer, dimension(:), intent(in) :: val
      integer, dimension(size(res)) :: tmp
    ⟨Sorting: merge⟩
    end subroutine merge_int

    subroutine merge_real (res, src1, src2, val)
      integer, dimension(:), intent(out) :: res
      integer, dimension(:), intent(in) :: src1, src2
      real(default), dimension(:), intent(in) :: val
      integer, dimension(size(res)) :: tmp
    ⟨Sorting: merge⟩
    end subroutine merge_real
```

⟨*Sorting: merge*⟩≡

```
    integer :: i1, i2, i
    i1 = 1
    i2 = 1
    do i = 1, size (tmp)
       if (val(src1(i1)) <= val(src2(i2))) then
          tmp(i) = src1(i1);   i1 = i1 + 1
          if (i1 > size (src1)) then
             tmp(i+1:) = src2(i2:)
             exit
          end if
       else
```

96

```
        tmp(i) = src2(i2);  i2 = i2 + 1
        if (i2 > size (src2)) then
           tmp(i+1:) = src1(i1:)
           exit
        end if
      end if
    end do
    res = tmp
```

2.13.2   Test

⟨*Sorting: public*⟩+≡
```
    public :: sorting_test
```

⟨*Sorting: procedures*⟩+≡
```
    subroutine sorting_test ()
      integer, parameter :: NMAX = 10
      real(default), dimension(NMAX) :: rval
      integer, dimension(NMAX) :: ival
      real, dimension(NMAX) :: harvest
      integer :: i
      print *, "Sorting real values:"
      do i = 1, NMAX
         print *
         call random_number (harvest(:i))
         rval(:i) = harvest(:i)
         print "(10(1x,F7.4))", rval(:i)
         rval(:i) = sort (rval(:i))
         print "(10(1x,F7.4))", rval(:i)
      end do
      print *
      print *, "Sorting integer values:"
      do i = 1, NMAX
         print *
         call random_number (harvest(:i))
         ival(:i) = harvest(:i) * NMAX * 2
         print "(10(1x,I2))", ival(:i)
         ival(:i) = sort (ival(:i))
         print "(10(1x,I2))", ival(:i)
      end do
    end subroutine sorting_test
```

Chapter 3

Text handling

WHIZARD has to handle complex structures in input (and output) data. Doing this in a generic and transparent way requires a generic lexer and parser. The necessary modules are implemented here:

ifiles Implementation of line-oriented internal files in a more flexible way (linked lists of variable-length strings) than the Fortran builtin features.

lexers Read text and transform it into a token stream.

syntax_rules Define the rules for interpreting tokens, to be used by the parser.

parser Categorize tokens (keyword, string, number etc.) and use a set of syntax rules to transform the input into a parse tree.

3.1 Internal files

The internal files introduced here (`ifile`) are a replacement for the built-in
internal files, which are fixed-size arrays of fixed-length character strings. The
`ifile` type is a doubly-linked list of variable-length character strings with line
numbers.

⟨`ifiles.f90`⟩≡
  ⟨*File header*⟩

```
  module ifiles
```

  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
    use limits, only: EOF !NODEP!
```

  ⟨*Standard module head*⟩

  ⟨*Ifiles: public*⟩

  ⟨*Ifiles: types*⟩

  ⟨*Ifiles: interfaces*⟩

```
  contains
```

  ⟨*Ifiles: subroutines*⟩

```
  end module ifiles
```

3.1.1 iostat codes

These are not specified in the standard, but apparently universal to current
compilers (checked for ifort, gfortran, g95, f95):

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: EOF = -1,  EOR = -2
```

3.1.2 The line type

The line entry type is for internal use, it is the list entry to be collected in an
`ifile` object.

⟨*Ifiles: types*⟩≡
```
    type :: line_entry_t
      private
      type(line_entry_t), pointer :: previous => null ()
      type(line_entry_t), pointer :: next => null ()
      type(string_t) :: string
      integer :: index
    end type line_entry_t
```

Create a new list entry, given a varying string as input. The line number and
pointers are not set, these make sense only within an `ifile`.

⟨*Ifiles: subroutines*⟩≡
```
subroutine line_entry_create (line, string)
  type(line_entry_t), pointer :: line
  type(string_t), intent(in) :: string
  allocate (line)
  line%string = string
end subroutine line_entry_create
```

Destroy a single list entry: Since the pointer components should not be deallo-
cated explicitly, just deallocate the object itself.

⟨*Ifiles: subroutines*⟩+≡
```
subroutine line_entry_destroy (line)
  type(line_entry_t), pointer :: line
  deallocate (line)
end subroutine line_entry_destroy
```

3.1.3 The ifile type

The internal file is a linked list of line entries.

⟨*Ifiles: public*⟩≡
```
public :: ifile_t
```

⟨*Ifiles: types*⟩+≡
```
type :: ifile_t
  private
  type(line_entry_t), pointer :: first => null ()
  type(line_entry_t), pointer :: last => null ()
  integer :: n_lines = 0
end type ifile_t
```

We need no explicit initializer, but a routine which recursively deallocates the
contents may be appropriate. After this, existing line pointers may become
undefined, so they should be nullified before the file is destroyed.

⟨*Ifiles: public*⟩+≡
```
public :: ifile_clear
```

⟨*Ifiles: subroutines*⟩+≡
```
subroutine ifile_clear (ifile)
  type(ifile_t), intent(inout) :: ifile
  type(line_entry_t), pointer :: current
  do while (associated (ifile%first))
     current => ifile%first
     ifile%first => current%next
     call line_entry_destroy (current)
  end do
  nullify (ifile%last)
  ifile%n_lines = 0
end subroutine ifile_clear
```

The finalizer is just an alias for the above.

⟨*Ifiles: public*⟩+≡
```
    public :: ifile_final
```

⟨*Ifiles: interfaces*⟩≡
```
    interface ifile_final
       module procedure ifile_clear
    end interface
```

3.1.4   I/O on ifiles

Fill an ifile from an ordinary external file, i.e., I/O unit. If the ifile is not empty, the old contents will be destroyed. We can read a fixed-length character string, an ISO varying string, an ordinary internal file (character-string array), or from an external unit. In the latter case, lines are appended until EOF is reached. Finally, there is a variant which reads from another ifile, effectively copying it.

⟨*Ifiles: public*⟩+≡
```
    public :: ifile_read
```

⟨*Ifiles: interfaces*⟩+≡
```
    interface ifile_read
       module procedure ifile_read_from_string
       module procedure ifile_read_from_char
       module procedure ifile_read_from_unit
       module procedure ifile_read_from_char_array
       module procedure ifile_read_from_ifile
    end interface
```

⟨*Ifiles: subroutines*⟩+≡
```
    subroutine ifile_read_from_string (ifile, string)
      type(ifile_t), intent(inout) :: ifile
      type(string_t), intent(in) :: string
      call ifile_clear (ifile)
      call ifile_append (ifile, string)
    end subroutine ifile_read_from_string

    subroutine ifile_read_from_char (ifile, char)
      type(ifile_t), intent(inout) :: ifile
      character(*), intent(in) :: char
      call ifile_clear (ifile)
      call ifile_append (ifile, char)
    end subroutine ifile_read_from_char

    subroutine ifile_read_from_char_array (ifile, char)
      type(ifile_t), intent(inout) :: ifile
      character(*), dimension(:), intent(in) :: char
      call ifile_clear (ifile)
      call ifile_append (ifile, char)
    end subroutine ifile_read_from_char_array

    subroutine ifile_read_from_unit (ifile, unit, iostat)
      type(ifile_t), intent(inout) :: ifile
      integer, intent(in) :: unit
      integer, intent(out), optional :: iostat
```

```
      call ifile_clear (ifile)
      call ifile_append (ifile, unit, iostat)
    end subroutine ifile_read_from_unit

    subroutine ifile_read_from_ifile (ifile, ifile_in)
      type(ifile_t), intent(inout) :: ifile
      type(ifile_t), intent(in) :: ifile_in
      call ifile_clear (ifile)
      call ifile_append (ifile, ifile_in)
    end subroutine ifile_read_from_ifile
```

Append to an ifile. The same as reading, but without resetting the ifile. In addition, there is a routine for appending a whole ifile.

⟨*Ifiles: public*⟩+≡
```
    public :: ifile_append
```

⟨*Ifiles: interfaces*⟩+≡
```
    interface ifile_append
       module procedure ifile_append_from_string
       module procedure ifile_append_from_char
       module procedure ifile_append_from_unit
       module procedure ifile_append_from_char_array
       module procedure ifile_append_from_ifile
    end interface
```

⟨*Ifiles: subroutines*⟩+≡
```
    subroutine ifile_append_from_string (ifile, string)
      type(ifile_t), intent(inout) :: ifile
      type(string_t), intent(in) :: string
      type(line_entry_t), pointer :: current
      call line_entry_create (current, string)
      current%index = ifile%n_lines + 1
      if (associated (ifile%last)) then
         current%previous => ifile%last
         ifile%last%next => current
      else
         ifile%first => current
      end if
      ifile%last => current
      ifile%n_lines = current%index
    end subroutine ifile_append_from_string

    subroutine ifile_append_from_char (ifile, char)
      type(ifile_t), intent(inout) :: ifile
      character(*), intent(in) :: char
      call ifile_append_from_string (ifile, var_str (trim (char)))
    end subroutine ifile_append_from_char

    subroutine ifile_append_from_char_array (ifile, char)
      type(ifile_t), intent(inout) :: ifile
      character(*), dimension(:), intent(in) :: char
      integer :: i
      do i = 1, size (char)
         call ifile_append_from_string (ifile, var_str (trim (char(i))))
      end do
```

```
    end subroutine ifile_append_from_char_array

    subroutine ifile_append_from_unit (ifile, unit, iostat)
      type(ifile_t), intent(inout) :: ifile
      integer, intent(in) :: unit
      integer, intent(out), optional :: iostat
      type(string_t) :: buffer
      integer :: ios
      ios = 0
      READ_LOOP: do
         call get (unit, buffer, iostat = ios)
         if (ios == EOF .or. ios > 0)  exit READ_LOOP
         call ifile_append_from_string (ifile, buffer)
      end do READ_LOOP
      if (present (iostat)) then
         iostat = ios
      else if (ios > 0) then
         call get (unit, buffer)  ! trigger error again
      end if
    end subroutine ifile_append_from_unit

    subroutine ifile_append_from_ifile (ifile, ifile_in)
      type(ifile_t), intent(inout) :: ifile
      type(ifile_t), intent(in) :: ifile_in
      type(line_entry_t), pointer :: current
      current => ifile_in%first
      do while (associated (current))
         call ifile_append_from_string (ifile, current%string)
         current => current%next
      end do
    end subroutine ifile_append_from_ifile
```

Write the ifile contents to an external unit

⟨*Ifiles: public*⟩+≡
```
    public :: ifile_write
```

⟨*Ifiles: subroutines*⟩+≡
```
    subroutine ifile_write (ifile, unit, iostat)
      type(ifile_t), intent(in) :: ifile
      integer, intent(in), optional :: unit
      integer, intent(out), optional :: iostat
      integer :: u
      type(line_entry_t), pointer :: current
      u = output_unit (unit);  if (u < 0)  return
      current => ifile%first
      do while (associated (current))
         call put_line (u, current%string, iostat)
         current => current%next
      end do
    end subroutine ifile_write
```

Convert the ifile to an array of strings, which is allocated by this function:

⟨*Ifiles: public*⟩+≡
```
    public :: ifile_to_string_array
```

⟨*Ifiles: subroutines*⟩+≡
```
    subroutine ifile_to_string_array (ifile, string)
      type(ifile_t), intent(in) :: ifile
      type(string_t), dimension(:), intent(inout), allocatable :: string
      type(line_entry_t), pointer :: current
      integer :: i
      allocate (string (ifile_get_length (ifile)))
      current => ifile%first
      do i = 1, ifile_get_length (ifile)
         string(i) = current%string
         current => current%next
      end do
    end subroutine ifile_to_string_array
```

3.1.5 Ifile tools

⟨*Ifiles: public*⟩+≡
```
    public :: ifile_get_length
```

⟨*Ifiles: subroutines*⟩+≡
```
    function ifile_get_length (ifile) result (length)
      integer :: length
      type(ifile_t), intent(in) :: ifile
      length = ifile%n_lines
    end function ifile_get_length
```

3.1.6 Line pointers

Instead of the implicit pointer used in ordinary file access, we define explicit
pointers, so there can be more than one at a time.

⟨*Ifiles: public*⟩+≡
```
    public :: line_p
```

⟨*Ifiles: types*⟩+≡
```
    type :: line_p
       private
       type(line_entry_t), pointer :: p => null ()
    end type line_p
```

Assign a file pointer to the first or last line in an ifile:

⟨*Ifiles: public*⟩+≡
```
    public :: line_init
```

⟨*Ifiles: subroutines*⟩+≡
```
    subroutine line_init (line, ifile, back)
      type(line_p), intent(inout) :: line
      type(ifile_t), intent(in) :: ifile
      logical, intent(in), optional :: back
      if (present (back)) then
         if (back) then
            line%p => ifile%last
         else
```

```
            line%p => ifile%first
        end if
    else
        line%p => ifile%first
    end if
end subroutine line_init
```

Remove the pointer association:

⟨*Ifiles: public*⟩+≡
```
    public :: line_final
```

⟨*Ifiles: subroutines*⟩+≡
```
    subroutine line_final (line)
      type(line_p), intent(inout) :: line
      nullify (line%p)
    end subroutine line_final
```

Go one step forward

⟨*Ifiles: public*⟩+≡
```
    public :: line_advance
```

⟨*Ifiles: subroutines*⟩+≡
```
    subroutine line_advance (line)
      type(line_p), intent(inout) :: line
      if (associated (line%p))  line%p => line%p%next
    end subroutine line_advance
```

Go one step backward

⟨*Ifiles: public*⟩+≡
```
    public :: line_backspace
```

⟨*Ifiles: subroutines*⟩+≡
```
    subroutine line_backspace (line)
      type(line_p), intent(inout) :: line
      if (associated (line%p))  line%p => line%p%previous
    end subroutine line_backspace
```

Check whether we are accessing a valid line

⟨*Ifiles: public*⟩+≡
```
    public :: line_is_associated
```

⟨*Ifiles: subroutines*⟩+≡
```
    function line_is_associated (line) result (ok)
      logical :: ok
      type(line_p), intent(in) :: line
      ok = associated (line%p)
    end function line_is_associated
```

3.1.7 Access lines via pointers

We do not need the ifile as an argument to these functions, because the `line` type will point to an existing ifile.

⟨*Ifiles: public*⟩+≡
```
    public :: line_get_string
```

⟨*Ifiles: subroutines*⟩+≡
```
    function line_get_string (line) result (string)
      type(string_t) :: string
      type(line_p), intent(in) :: line
      if (associated (line%p)) then
         string = line%p%string
      else
         string = ""
      end if
    end function line_get_string
```

Variant where the line pointer is advanced after reading.

⟨*Ifiles: public*⟩+≡
```
    public :: line_get_string_advance
```

⟨*Ifiles: subroutines*⟩+≡
```
    function line_get_string_advance (line) result (string)
      type(string_t) :: string
      type(line_p), intent(inout) :: line
      if (associated (line%p)) then
         string = line%p%string
         call line_advance (line)
      else
         string = ""
      end if
    end function line_get_string_advance
```

⟨*Ifiles: public*⟩+≡
```
    public :: line_get_index
```

⟨*Ifiles: subroutines*⟩+≡
```
    function line_get_index (line) result (index)
      integer :: index
      type(line_p), intent(in) :: line
      if (associated (line%p)) then
         index = line%p%index
      else
         index = 0
      end if
    end function line_get_index
```

⟨*Ifiles: public*⟩+≡
```
    public :: line_get_length
```

⟨*Ifiles: subroutines*⟩+≡
```
    function line_get_length (line) result (length)
      integer :: length
      type(line_p), intent(in) :: line
```

```
      if (associated (line%p)) then
         length = len (line%p%string)
      else
         length = 0
      end if
end function line_get_length
```

3.2 Lexer

The lexer purpose is to read from a line-separated character input stream (usually a file) and properly chop the stream into lexemes (tokens). [The parser will transform lexemes into meaningful tokens, to be stored in a parse tree, therefore we do not use the term 'token' here.] The input is read line-by-line, but interpreted free-form, except for quotes and the comment syntax. (Fortran 2003 would allow us to use a stream type for reading.)

In an object-oriented approach, we can dynamically create and destroy lexers, including the lexer setup.

The main lexer function is to return a lexeme according to the basic lexer rules (quotes, comments, whitespace, special classes). There is also a routine to write back a lexeme to the input stream (but only once).

For the rules, we separate the possible characters into classes. Whitespace usually consists of blank, tab, and line-feed, where any number of consecutive whitespace is equivalent to one. Quoted strings are enclosed by a pair of quote characters, possibly multiline. Comments are similar to quotes, but interpreted as whitespace. Numbers are identified (not distinguishing real and integer) but not interpreted. Other character classes make up identifiers.

⟨`lexers.f90`⟩≡
  ⟨*File header*⟩

  module lexers

  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use limits, only: EOF, EOR !NODEP!
    use limits, only: LF !NODEP!
    use limits, only: WHITESPACE_CHARS, LCLETTERS, UCLETTERS, DIGITS !NODEP!
    use diagnostics !NODEP!
    use ifiles, only: ifile_t
    use ifiles, only: line_p, line_is_associated, line_init, line_final
    use ifiles, only: line_get_string_advance

  ⟨*Standard module head*⟩

  ⟨*Lexer: public*⟩

  ⟨*Lexer: parameters*⟩

  ⟨*Lexer: types*⟩

  ⟨*Lexer: interfaces*⟩

  contains

  ⟨*Lexer: procedures*⟩

  end module lexers

3.2.1 Input streams

For flexible input, we define a generic stream type that refers to either an external file, an external unit which is already open, a string, an `ifile` object (internal file, i.e., string list), or a line pointer to an `ifile` object. The stream type actually follows the idea of a formatted external file, which is line-oriented. Thus, the stream reader always returns a whole record (input line).

  Note that only in the string version, the stream contents are stored inside the stream object. In the `ifile` version, the stream contains only the line pointer, while in the external-file case, the line pointer is implicitly created by the runtime library.

⟨*Lexer: public*⟩≡
```
    public :: stream_t
```

⟨*Lexer: types*⟩≡
```
    type :: stream_t
       type(string_t), pointer :: filename => null ()
       integer, pointer :: unit => null ()
       type(string_t), pointer :: string => null ()
       type(ifile_t), pointer :: ifile => null ()
       type(line_p), pointer :: line => null ()
       integer :: record = 0
       logical :: eof = .false.
    end type stream_t
```

The initializers refer to the specific version. The stream should be undefined before calling this.

⟨*Lexer: public*⟩+≡
```
    public :: stream_init
```

⟨*Lexer: interfaces*⟩≡
```
    interface stream_init
       module procedure stream_init_filename
       module procedure stream_init_unit
       module procedure stream_init_string
       module procedure stream_init_ifile
       module procedure stream_init_line
    end interface
```

⟨*Lexer: procedures*⟩≡
```
    subroutine stream_init_filename (stream, filename)
      type(stream_t), intent(out) :: stream
      character(*), intent(in) :: filename
      integer :: unit
      unit = free_unit ()
      open (unit=unit, file=filename, status="old", action="read")
      call stream_init_unit (stream, unit)
      allocate (stream%filename)
      stream%filename = filename
    end subroutine stream_init_filename

    subroutine stream_init_unit (stream, unit)
      type(stream_t), intent(out) :: stream
      integer, intent(in) :: unit
```

```
    allocate (stream%unit)
    stream%unit = unit
    stream%eof = .false.
  end subroutine stream_init_unit

  subroutine stream_init_string (stream, string)
    type(stream_t), intent(out) :: stream
    type(string_t), intent(in) :: string
    allocate (stream%string)
    stream%string = string
  end subroutine stream_init_string

  subroutine stream_init_ifile (stream, ifile)
    type(stream_t), intent(out) :: stream
    type(ifile_t), intent(in) :: ifile
    type(line_p) :: line
    call line_init (line, ifile)
    call stream_init_line (stream, line)
    allocate (stream%ifile)
    stream%ifile = ifile
  end subroutine stream_init_ifile

  subroutine stream_init_line (stream, line)
    type(stream_t), intent(out) :: stream
    type(line_p), intent(in) :: line
    allocate (stream%line)
    stream%line = line
  end subroutine stream_init_line
```

The finalizer restores the initial state. If an external file was opened, it is closed.

⟨*Lexer: public*⟩+≡
```
    public :: stream_final
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine stream_final (stream)
      type(stream_t), intent(inout) :: stream
      if (associated (stream%filename)) then
         close (stream%unit)
         deallocate (stream%unit)
         deallocate (stream%filename)
      else if (associated (stream%unit)) then
         deallocate (stream%unit)
      else if (associated (stream%string)) then
         deallocate (stream%string)
      else if (associated (stream%ifile)) then
         call line_final (stream%line)
         deallocate (stream%line)
         deallocate (stream%ifile)
      else if (associated (stream%line)) then
         call line_final (stream%line)
         deallocate (stream%line)
      end if
    end subroutine stream_final
```

This returns the next record from the input stream. Depending on the stream type, the stream pointers are modified: Reading from external unit, the external file is advanced (implicitly). Reading from string, the string is replaced by an empty string. Reading from `ifile`, the line pointer is advanced. Note that the `iostat` argument is mandatory.

⟨*Lexer: public*⟩+≡

```
public :: stream_get_record
```

⟨*Lexer: procedures*⟩+≡

```
subroutine stream_get_record (stream, string, iostat)
  type(stream_t), intent(inout) :: stream
  type(string_t), intent(out) :: string
  integer, intent(out) :: iostat
  if (associated (stream%unit)) then
     if (stream%eof) then
        iostat = EOF
     else
        call get (stream%unit, string, iostat=iostat)
        if (iostat == EOR)  iostat = 0
        if (iostat == EOF) then
           iostat = 0
           stream%eof = .true.
        else if (iostat == 0) then
           stream%record = stream%record + 1
        end if
     end if
  else if (associated (stream%string)) then
     if (len (stream%string) /= 0) then
        string = stream%string
        stream%string = ""
        iostat = 0
        stream%record = stream%record + 1
     else
        string = ""
        iostat = EOF
     end if
  else if (associated (stream%line)) then
     if (line_is_associated (stream%line)) then
        string = line_get_string_advance (stream%line)
        iostat = 0
        stream%record = stream%record + 1
     else
        string = ""
        iostat = EOF
     end if
  else
     call msg_bug (" Attempt to read from uninitialized input stream")
  end if
end subroutine stream_get_record
```

Return the current stream source as a message string.

⟨*Lexer: public*⟩+≡

```
public :: stream_get_source_info_string
```

⟨Lexer: procedures⟩+≡

```
function stream_get_source_info_string (stream) result (string)
  type(string_t) :: string
  type(stream_t), intent(in) :: stream
  character(20) :: buffer
  if (associated (stream%filename)) then
     string = "File '" // stream%filename // "' (unit = "
     write (buffer, "(I0)")  stream%unit
     string = string // trim (buffer) // ")"
  else if (associated (stream%unit)) then
     write (buffer, "(I0)")  stream%unit
     string = "Unit " // trim (buffer)
  else if (associated (stream%string)) then
     string = "Input string"
  else if (associated (stream%ifile) .or. associated (stream%line)) then
     string = "Internal file"
  else
     string = ""
  end if
end function stream_get_source_info_string
```

Return the index of the record just read as a message string.

⟨Lexer: public⟩+≡

```
public :: stream_get_record_info_string
```

⟨Lexer: procedures⟩+≡

```
function stream_get_record_info_string (stream) result (string)
  type(string_t) :: string
  type(stream_t), intent(in) :: stream
  character(20) :: buffer
  string = stream_get_source_info_string (stream)
  if (string /= "")  string = string // ", "
  write (buffer, "(I0)")  stream%record
  string = string // "line " // trim (buffer)
end function stream_get_record_info_string
```

3.2.2 Keyword list

The lexer should be capable of identifying a token as a known keyword. To this
end, we store a list of keywords:

⟨Lexer: public⟩+≡

```
public :: keyword_list_t
```

⟨Lexer: types⟩+≡

```
type :: keyword_entry_t
  private
  type(string_t) :: string
  type(keyword_entry_t), pointer :: next => null ()
end type keyword_entry_t

type :: keyword_list_t
  private
  type(keyword_entry_t), pointer :: first => null ()
```

```
      type(keyword_entry_t), pointer :: last => null ()
   end type keyword_list_t
```

Add a new string to the keyword list, unless it is already there:

⟨*Lexer: public*⟩+≡
```
   public :: keyword_list_add
```

⟨*Lexer: procedures*⟩+≡
```
   subroutine keyword_list_add (keylist, string)
     type(keyword_list_t), intent(inout) :: keylist
     type(string_t), intent(in) :: string
     type(keyword_entry_t), pointer :: k_entry_new
     if (.not. keyword_list_contains (keylist, string)) then
        allocate (k_entry_new)
        k_entry_new%string = string
        if (associated (keylist%first)) then
           keylist%last%next => k_entry_new
        else
           keylist%first => k_entry_new
        end if
        keylist%last => k_entry_new
     end if
   end subroutine keyword_list_add
```

Return true if a string is a keyword.

⟨*Lexer: public*⟩+≡
```
   public :: keyword_list_contains
```

⟨*Lexer: procedures*⟩+≡
```
   function keyword_list_contains (keylist, string) result (found)
     type(keyword_list_t), intent(in) :: keylist
     type(string_t), intent(in) :: string
     logical :: found
     found = .false.
     call check_rec (keylist%first)
   contains
     recursive subroutine check_rec (k_entry)
       type(keyword_entry_t), pointer :: k_entry
       if (associated (k_entry)) then
          if (k_entry%string /= string) then
             call check_rec (k_entry%next)
          else
             found = .true.
          end if
       end if
     end subroutine check_rec
   end function keyword_list_contains
```

Write the keyword list

⟨*Lexer: public*⟩+≡
```
   public :: keyword_list_write
```

⟨Lexer: interfaces⟩+≡

```
    interface keyword_list_write
       module procedure keyword_list_write_unit
    end interface
```

⟨Lexer: procedures⟩+≡

```
    subroutine keyword_list_write_unit (keylist, unit)
      type(keyword_list_t), intent(in) :: keylist
      integer, intent(in) :: unit
      write (unit, "(A)") "Keyword list:"
      if (associated (keylist%first)) then
         call keyword_write_rec (keylist%first)
         write (unit, *)
      else
         write (unit, "(1x,A)") "[empty]"
      end if
    contains
      recursive subroutine keyword_write_rec (k_entry)
        type(keyword_entry_t), intent(in), pointer :: k_entry
        if (associated (k_entry)) then
           write (unit, "(1x,A)", advance="no")  char (k_entry%string)
           call keyword_write_rec (k_entry%next)
        end if
      end subroutine keyword_write_rec
    end subroutine keyword_list_write_unit
```

Clear the keyword list

⟨Lexer: public⟩+≡

```
    public :: keyword_list_final
```

⟨Lexer: procedures⟩+≡

```
    subroutine keyword_list_final (keylist)
      type(keyword_list_t), intent(inout) :: keylist
      call keyword_destroy_rec (keylist%first)
      nullify (keylist%last)
    contains
      recursive subroutine keyword_destroy_rec (k_entry)
        type(keyword_entry_t), pointer :: k_entry
        if (associated (k_entry)) then
           call keyword_destroy_rec (k_entry%next)
           deallocate (k_entry)
        end if
      end subroutine keyword_destroy_rec
    end subroutine keyword_list_final
```

3.2.3   Lexeme templates

This type is handled like a rudimentary regular expression. It determines the lexer behavior when matching a string. The actual objects made from this type and the corresponding matching routines are listed below.

⟨Lexer: types⟩+≡

```
    type :: template_t
```

```
      private
      integer :: type
      character(256) :: charset1, charset2
      integer :: len1, len2
    end type template_t
```

These are the types that valid lexemes can have:

⟨*Lexer: public*⟩+≡
```
    public :: T_KEYWORD, T_IDENTIFIER, T_QUOTED, T_NUMERIC
```

⟨*Lexer: parameters*⟩≡
```
    integer, parameter :: T_KEYWORD = 1
    integer, parameter :: T_IDENTIFIER = 2, T_QUOTED = 3, T_NUMERIC = 4
```

These are special types:

⟨*Lexer: parameters*⟩+≡
```
    integer, parameter :: EMPTY = 0, WHITESPACE = 10
    integer, parameter :: NO_MATCH = 11, IO_ERROR = 12, OVERFLOW = 13
    integer, parameter :: UNMATCHED_QUOTE = 14
```

In addition, we have EOF which is a negative integer, normally −1. Printout for debugging:

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexeme_type_write (type, unit)
      integer, intent(in) :: type
      integer, intent(in) :: unit
      select case (type)
      case (EMPTY);        write(unit,"(A)",advance="no") " EMPTY      "
      case (WHITESPACE);   write(unit,"(A)",advance="no") " WHITESPACE "
      case (T_IDENTIFIER); write(unit,"(A)",advance="no") " IDENTIFIER "
      case (T_QUOTED);     write(unit,"(A)",advance="no") " QUOTED     "
      case (T_NUMERIC);    write(unit,"(A)",advance="no") " NUMERIC    "
      case (IO_ERROR);     write(unit,"(A)",advance="no") " IO_ERROR   "
      case (OVERFLOW);     write(unit,"(A)",advance="no") " OVERFLOW   "
      case (UNMATCHED_QUOTE);   write(unit,"(A)",advance="no") " UNMATCHEDQ "
      case (NO_MATCH);     write(unit,"(A)",advance="no") " NO_MATCH   "
      case (EOF);          write(unit,"(A)",advance="no") " EOF        "
      case default;        write(unit,"(A)",advance="no") " [illegal]  "
      end select
    end subroutine lexeme_type_write

    subroutine template_write (tt, unit)
      type(template_t), intent(in) :: tt
      integer, intent(in) :: unit
      call lexeme_type_write (tt%type, unit)
      write (unit, "(A)", advance="no") "'" // tt%charset1(1:tt%len1) // "'"
      write (unit, "(A)", advance="no") " '" // tt%charset2(1:tt%len2) // "'"
    end subroutine template_write
```

The matching functions all return the number of matched characters in the provided string. If this number is zero, the match has failed.

The `template` functions are declared `pure` because they appear in `forall` loops below.

A template for whitespace:

⟨*Lexer: procedures*⟩+≡

```
pure function template_whitespace (chars) result (tt)
  character(*), intent(in) :: chars
  type(template_t) :: tt
  tt = template_t (WHITESPACE, chars, "", len (chars), 0)
end function template_whitespace
```

Just match the string against the character set.

⟨*Lexer: procedures*⟩+≡

```
subroutine match_whitespace (tt, s, n)
  type(template_t), intent(in) :: tt
  character(*), intent(in) :: s
  integer, intent(out) :: n
  n = verify (s, tt%charset1(1:tt%len1)) - 1
  if (n < 0)  n = len (s)
end subroutine match_whitespace
```

A template for normal identifiers. To match, a lexeme should have a first character in class **chars1** and an arbitrary number of further characters in class **chars2**. If the latter is empty, we are looking for a single-character lexeme.

⟨*Lexer: procedures*⟩+≡

```
pure function template_identifier (chars1, chars2) result (tt)
  character(*), intent(in) :: chars1, chars2
  type(template_t) :: tt
  tt = template_t (T_IDENTIFIER, chars1, chars2, len(chars1), len(chars2))
end function template_identifier
```

Here, the first letter must match, the others may or may not.

⟨*Lexer: procedures*⟩+≡

```
subroutine match_identifier (tt, s, n)
  type(template_t), intent(in) :: tt
  character(*), intent(in) :: s
  integer, intent(out) :: n
  if (verify (s(1:1), tt%charset1(1:tt%len1)) == 0) then
     n = verify (s(2:), tt%charset2(1:tt%len2))
     if (n == 0)  n = len (s)
  else
     n = 0
  end if
end subroutine match_identifier
```

A template for quoted strings. The same template applies for comments. The first character set indicates the left quote (could be a sequence of several characters), the second one the matching right quote.

⟨*Lexer: procedures*⟩+≡

```
pure function template_quoted (chars1, chars2) result (tt)
  character(*), intent(in) :: chars1, chars2
  type(template_t) :: tt
  tt = template_t (T_QUOTED, chars1, chars2, len (chars1), len (chars2))
end function template_quoted
```

116

Here, the beginning of the string must exactly match the first character set, then we look for the second one. If found, return. If there is a first quote but no second one, return a negative number, indicating this error condition.

⟨*Lexer: procedures*⟩+≡

```
subroutine match_quoted (tt, s, n, range)
  type(template_t), intent(in) :: tt
  character(*), intent(in) :: s
  integer, intent(out) :: n
  integer, dimension(2), intent(out) :: range
  character(tt%len1) :: ch1
  character(tt%len2) :: ch2
  integer :: i
  ch1 = tt%charset1
  if (s(1:tt%len1) == ch1) then
     ch2 = tt%charset2
     do i = tt%len1 + 1, len (s) - tt%len2 + 1
        if (s(i:i+tt%len2-1) == ch2) then
           n = i + tt%len2 - 1
           range(1) = tt%len1 + 1
           range(2) = i - 1
           return
        end if
     end do
     n = -1
     range = 0
  else
     n = 0
     range = 0
  end if
end subroutine match_quoted
```

A template for real numbers. The first character set is the set of allowed exponent letters. In accordance with the other functions we return the lexeme as a string but do not read it.

⟨*Lexer: procedures*⟩+≡

```
pure function template_numeric (chars) result (tt)
  character(*), intent(in) :: chars
  type(template_t) :: tt
  tt = template_t (T_NUMERIC, chars, "", len (chars), 0)
end function template_numeric
```

A numeric lexeme may be real or integer. We purposely do not allow for a preceding sign. If the number is followed by an exponent, this is included, otherwise the rest is ignored.

There is a possible pitfall with this behavior: while the string `1e3` will be interpreted as a single number, the analogous string `1a3` will be split into the number 1 and an identifier `a3`. There is no easy way around such an ambiguity. We should make sure that the syntax does not contain identifiers like `a3` or `e3`.

⟨*Lexer: procedures*⟩+≡

```
subroutine match_numeric (tt, s, n)
  type(template_t), intent(in) :: tt
  character(*), intent(in) :: s
```

```
      integer, intent(out) :: n
      integer :: i, n0
      character(10), parameter :: digits = "0123456789"
      character(2), parameter :: signs = "-+"
      n = verify (s, digits) - 1
      if (n < 0) then
         n = 0
         return
      else if (s(n+1:n+1) == ".") then
         i = verify (s(n+2:), digits) - 1
         if (i < 0) then
            n = len (s)
            return
         else if (i > 0 .or. n > 0) then
            n = n + 1 + i
         end if
      end if
      n0 = n
      if (n > 0) then
         if (verify (s(n+1:n+1), tt%charset1(1:tt%len1)) == 0) then
            n = n + 1
            if (verify (s(n+1:n+1), signs) == 0)  n = n + 1
            i = verify (s(n+1:), digits) - 1
            if (i < 0) then
               n = len (s)
            else if (i == 0) then
               n = n0
            else
               n = n + i
            end if
         end if
      end if
   end subroutine match_numeric
```

The generic matching routine. With Fortran 2003 we would define separate types and use a SELECT TYPE instead.

⟨*Lexer: procedures*⟩+≡

```
   subroutine match_template (tt, s, n, range)
     type(template_t), intent(in) :: tt
     character(*), intent(in) :: s
     integer, intent(out) :: n
     integer, dimension(2), intent(out) :: range
     select case (tt%type)
     case (WHITESPACE)
        call match_whitespace (tt, s, n)
        range = 0
     case (T_IDENTIFIER)
        call match_identifier (tt, s, n)
        range(1) = 1
        range(2) = len_trim (s)
     case (T_QUOTED)
        call match_quoted (tt, s, n, range)
     case (T_NUMERIC)
```

```
          call match_numeric (tt, s, n)
          range(1) = 1
          range(2) = len_trim (s)
      case default
          call msg_bug ("Invalid lexeme template encountered")
      end select
   end subroutine match_template
```

Match against an array of templates. Return the index of the first template
that matches together with the number of characters matched and the range of
the relevant substring. If all fails, these numbers are zero.

⟨*Lexer: procedures*⟩+≡
```
   subroutine match (tt, s, n, range, ii)
     type(template_t), dimension(:), intent(in) :: tt
     character(*), intent(in) :: s
     integer, intent(out) :: n
     integer, dimension(2), intent(out) :: range
     integer, intent(out) :: ii
     integer :: i
     do i = 1, size (tt)
        call match_template (tt(i), s, n, range)
        if (n /= 0) then
           ii = i
           return
        end if
     end do
     n = 0
     ii = 0
   end subroutine match
```

3.2.4 The lexer setup

This object contains information about character classes. As said above, one
class consists of quoting chars (matching left and right), another one of com-
ment chars (similar), a class of whitespace, and several classes of characters
that make up identifiers. When creating the lexer setup, the character classes
are transformed into lexeme templates which are to be matched in a certain
predefined order against the input stream.

BLANK should always be taken as whitespace, some things may depend on
this. TAB is also fixed by convention, but may in principle be modified. Newline
(DOS!) and linefeed are also defined as whitespace.

⟨*Limits: public parameters*⟩+≡
```
   character, parameter, public :: BLANK = ' ',  TAB = achar(9)
   character, parameter, public :: CR = achar(13), LF = achar(10)
   character, parameter, public :: BACKSLASH = achar(92)
   character(*), parameter, public :: WHITESPACE_CHARS = BLANK// TAB // CR // LF
   character(*), parameter, public :: LCLETTERS = "abcdefghijklmnopqrstuvwxyz"
   character(*), parameter, public :: UCLETTERS = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
   character(*), parameter, public :: DIGITS = "0123456789"
```

The lexer setup, containing the list of lexeme templates. No defaults yet. The
type with index zero will be assigned to the `NO_MATCH` lexeme.

The keyword list is not stored, just a pointer to it. We anticipate that
the keyword list is part of the syntax table, and the lexer needs not alter it.
Furthermore, the lexer is typically finished before the syntax table is.

⟨*Lexer: parameters*⟩+≡
```
    integer, parameter :: CASE_KEEP = 0, CASE_UP = 1, CASE_DOWN = 2
```

⟨*Lexer: types*⟩+≡
```
    type :: lexer_setup_t
       private
       type(template_t), dimension(:), allocatable :: tt
       integer, dimension(:), allocatable :: type
       integer :: keyword_case = CASE_KEEP
       type(keyword_list_t), pointer :: keyword_list => null ()
    end type lexer_setup_t
```

Fill the lexer setup object. Some things are hardcoded here (whitespace, al-
phanumeric identifiers), some are free: comment chars (but these must be sin-
gle, and comments must be terminated by line-feed), quote chars and matches
(must be single), characters to be read as one-character lexeme, special classes
(characters of one class that should be glued together as identifiers).

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexer_setup_init (setup, &
         comment_chars, quote_chars, quote_match, &
         single_chars, special_class, &
         keyword_list, upper_case_keywords)
      type(lexer_setup_t), intent(inout) :: setup
      character(*), intent(in) :: comment_chars
      character(*), intent(in) :: quote_chars, quote_match
      character(*), intent(in) :: single_chars
      character(*), dimension(:), intent(in) :: special_class
      type(keyword_list_t), pointer :: keyword_list
      logical, intent(in), optional :: upper_case_keywords
      integer :: n, i
      if (present (upper_case_keywords)) then
         if (upper_case_keywords) then
            setup%keyword_case = CASE_UP
         else
            setup%keyword_case = CASE_DOWN
         end if
      else
         setup%keyword_case = CASE_KEEP
      end if
      n = 1 + len (comment_chars) + len (quote_chars) + 1 &
           + len (single_chars) + size (special_class) + 1
      allocate (setup%tt(n))
      allocate (setup%type(0:n))
      n = 0
      setup%type(n) = NO_MATCH
      n = n + 1
      setup%tt(n) = template_whitespace (WHITESPACE_CHARS)
```

```
      setup%type(n) = EMPTY
      forall (i = 1:len(comment_chars))
          setup%tt(n+i) = template_quoted (comment_chars(i:i), LF)
          setup%type(n+i) = EMPTY
      end forall
      n = n + len (comment_chars)
      forall (i = 1:len(quote_chars))
          setup%tt(n+i) = template_quoted (quote_chars(i:i), quote_match(i:i))
          setup%type(n+i) = T_QUOTED
      end forall
      n = n + len (quote_chars)
      setup%tt(n+1) = template_numeric ("EeDd")
      setup%type(n+1) = T_NUMERIC
      n = n + 1
      forall (i = 1:len (single_chars))
          setup%tt(n+i) = template_identifier (single_chars(i:i), "")
          setup%type(n+i) = T_IDENTIFIER
      end forall
      n = n + len (single_chars)
      forall (i = 1:size (special_class))
          setup%tt(n+i) = template_identifier &
              (trim (special_class(i)), trim (special_class(i)))
          setup%type(n+i) = T_IDENTIFIER
      end forall
      n = n + size (special_class)
      setup%tt(n+1) = template_identifier &
          (LCLETTERS//UCLETTERS, LCLETTERS//DIGITS//"_"//UCLETTERS)
      setup%type(n+1) = T_IDENTIFIER
      n = n + 1
      if (n /= size (setup%tt)) &
          call msg_bug ("Size mismatch in lexer setup")
      setup%keyword_list => keyword_list
    end subroutine lexer_setup_init
```

The destructor is needed only if the object is not itself part of an allocatable array

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexer_setup_final (setup)
      type(lexer_setup_t), intent(inout) :: setup
      deallocate (setup%tt, setup%type)
      setup%keyword_list => null ()
    end subroutine lexer_setup_final
```

For debugging: Write the lexer setup

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexer_setup_write (setup, unit)
      type(lexer_setup_t), intent(in) :: setup
      integer, intent(in) :: unit
      integer :: i
      write (unit, "(A)") "Lexer setup:"
      if (allocated (setup%tt)) then
          do i = 1, size (setup%tt)
              call template_write (setup%tt(i), unit)
```

```
         write (unit, '(A)', advance = "no")  " -> "
         call lexeme_type_write (setup%type(i), unit)
         write (unit, *)
      end do
   else
      write (unit, *) "[empty]"
   end if
   if (associated (setup%keyword_list)) then
      call keyword_list_write (setup%keyword_list, unit)
   end if
 end subroutine lexer_setup_write
```

3.2.5   The lexeme type

An object of this type is returned by the lexer. Apart from the lexeme string, it gives information about the relevant substring (first and last character index) and the lexeme type. Interpreting the string is up to the parser.

⟨*Lexer: public*⟩+≡
```
    public :: lexeme_t
```

⟨*Lexer: types*⟩+≡
```
    type :: lexeme_t
      private
      integer :: type = EMPTY
      type(string_t) :: s
      integer :: b = 0, e = 0
    end type lexeme_t
```

Debugging aid:

⟨*Lexer: public*⟩+≡
```
    public :: lexeme_write
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexeme_write (t, unit)
      type(lexeme_t), intent(in) :: t
      integer, intent(in) :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      select case (t%type)
      case (T_KEYWORD)
         write (u, *) "KEYWORD:    '" // char (t%s) // "'"
      case (T_IDENTIFIER)
         write (u, *) "IDENTIFIER: '" // char (t%s) // "'"
      case (T_QUOTED)
         write (u, *) "QUOTED:     '" // char (t%s) // "'"
      case (T_NUMERIC)
         write (u, *) "NUMERIC:    '" // char (t%s) // "'"
      case (UNMATCHED_QUOTE)
         write (u, *) "Unmatched quote: "// char (t%s)
      case (OVERFLOW); write (u, *) "Overflow: "// char (t%s)
      case (EMPTY);    write (u, *) "Empty lexeme"
      case (NO_MATCH); write (u, *) "No match"
      case (IO_ERROR); write (u, *) "IO error"
```

```
      case (EOF);       write (u, *) "EOF"
      case default
         write (u, *) "Error"
      end select
   end subroutine lexeme_write
```

Store string and type in a lexeme. The range determines the beginning and end
of the relevant part of the string. Check for a keyword.

⟨*Lexer: procedures*⟩+≡
```
   subroutine lexeme_set (t, keyword_list, s, range, type, keyword_case)
     type(lexeme_t), intent(out) :: t
     type(keyword_list_t), pointer :: keyword_list
     type(string_t), intent(in) :: s
     type(string_t) :: keyword
     integer, dimension(2), intent(in) :: range
     integer, intent(in) :: type
     integer, intent(in), optional :: keyword_case
     t%type = type
     if (present (keyword_case)) then
        select case (keyword_case)
        case (CASE_KEEP);   keyword = s
        case (CASE_UP);     keyword = upper_case (s)
        case (CASE_DOWN);   keyword = lower_case (s)
        end select
     else
        keyword = s
     end if
     if (type == T_IDENTIFIER) then
        if (associated (keyword_list)) then
           if (keyword_list_contains (keyword_list, keyword)) &
                t%type = T_KEYWORD
        end if
     end if
     select case (t%type)
     case (T_KEYWORD);  t%s = keyword
     case default;      t%s = s
     end select
     t%b = range(1)
     t%e = range(2)
   end subroutine lexeme_set

   subroutine lexeme_clear (t)
     type(lexeme_t), intent(out) :: t
     t%type = EMPTY
     t%s = ""
   end subroutine lexeme_clear
```

Retrieve the lexeme string, the relevant part of it, and the type. The last
function returns true if there is a break condition reached (error or EOF).

⟨*Lexer: public*⟩+≡
```
   public :: lexeme_get_string
   public :: lexeme_get_contents
   public :: lexeme_get_delimiters
```

```
    public :: lexeme_get_type
```

⟨*Lexer: procedures*⟩+≡
```
    function lexeme_get_string (t) result (s)
      type(string_t) :: s
      type(lexeme_t), intent(in) :: t
      s = t%s
    end function lexeme_get_string

    function lexeme_get_contents (t) result (s)
      type(string_t) :: s
      type(lexeme_t), intent(in) :: t
      s = extract (t%s, t%b, t%e)
    end function lexeme_get_contents

    function lexeme_get_delimiters (t) result (del)
      type(string_t), dimension(2) :: del
      type(lexeme_t), intent(in) :: t
      del(1) = extract (t%s, finish = t%b-1)
      del(2) = extract (t%s, start = t%e+1)
    end function lexeme_get_delimiters

    function lexeme_get_type (t) result (type)
      integer :: type
      type(lexeme_t), intent(in) :: t
      type = t%type
    end function lexeme_get_type
```

Check for a generic break condition (error/eof) and for eof in particular.

⟨*Lexer: public*⟩+≡
```
    public :: lexeme_is_break
    public :: lexeme_is_eof
```

⟨*Lexer: procedures*⟩+≡
```
    function lexeme_is_break (t) result (break)
      logical :: break
      type(lexeme_t), intent(in) :: t
      select case (t%type)
      case (EOF, IO_ERROR, OVERFLOW, NO_MATCH)
         break = .true.
      case default
         break = .false.
      end select
    end function lexeme_is_break

    function lexeme_is_eof (t) result (ok)
      logical :: ok
      type(lexeme_t), intent(in) :: t
      ok = t%type == EOF
    end function lexeme_is_eof
```

3.2.6 The lexer object

We store the current lexeme and the current line. The line buffer is set each time a new line is read from file. The working buffer has one character more, to hold any trailing blank. Pointers to line and column are for debugging, they will be used to make up readable error messages for the parser.

⟨*Lexer: public*⟩+≡
```
public :: lexer_t
```

⟨*Lexer: types*⟩+≡
```
type :: lexer_t
   private
   type(lexer_setup_t) :: setup
   type(stream_t), pointer :: stream => null ()
   type(lexeme_t) :: lexeme
   type(string_t) :: previous_line2
   type(string_t) :: previous_line1
   type(string_t) :: current_line
   integer :: lines_read = 0
   integer :: current_column = 0
   integer :: previous_column = 0
   type(string_t) :: buffer
   type(lexer_t), pointer :: parent => null ()
end type lexer_t
```

Create-setup wrapper

⟨*Lexer: public*⟩+≡
```
public :: lexer_init
```

⟨*Lexer: procedures*⟩+≡
```
subroutine lexer_init (lexer, &
     comment_chars, quote_chars, quote_match, &
     single_chars, special_class, &
     keyword_list, upper_case_keywords, &
     parent)
  type(lexer_t), intent(inout) :: lexer
  character(*), intent(in) :: comment_chars
  character(*), intent(in) :: quote_chars, quote_match
  character(*), intent(in) :: single_chars
  character(*), dimension(:), intent(in) :: special_class
  type(keyword_list_t), pointer :: keyword_list
  logical, intent(in), optional :: upper_case_keywords
  type(lexer_t), target, intent(in), optional :: parent
  call lexer_setup_init (lexer%setup, &
       comment_chars = comment_chars, &
       quote_chars = quote_chars, &
       quote_match = quote_match, &
       single_chars = single_chars, &
       special_class = special_class, &
       keyword_list = keyword_list, &
       upper_case_keywords = upper_case_keywords)
  if (present (parent))  lexer%parent => parent
  call lexer_clear (lexer)
end subroutine lexer_init
```

Clear the lexer state, but not the setup. This should be done when the lexing starts, but it is not known whether the lexer was used before.

⟨*Lexer: public*⟩+≡

```
public :: lexer_clear
```

⟨*Lexer: procedures*⟩+≡

```
subroutine lexer_clear (lexer)
  type(lexer_t), intent(inout) :: lexer
  call lexeme_clear (lexer%lexeme)
  lexer%previous_line2 = ""
  lexer%previous_line1 = ""
  lexer%current_line = ""
  lexer%lines_read = 0
  lexer%current_column = 0
  lexer%previous_column = 0
  lexer%buffer = ""
end subroutine lexer_clear
```

Reset lexer state and delete setup

⟨*Lexer: public*⟩+≡

```
public :: lexer_final
```

⟨*Lexer: procedures*⟩+≡

```
subroutine lexer_final (lexer)
  type(lexer_t), intent(inout) :: lexer
  call lexer_clear (lexer)
  call lexer_setup_final (lexer%setup)
end subroutine lexer_final
```

3.2.7 The lexer routine

For lexing we need to associate an input stream to the lexer.

⟨*Lexer: public*⟩+≡

```
public :: lexer_assign_stream
```

⟨*Lexer: procedures*⟩+≡

```
subroutine lexer_assign_stream (lexer, stream)
  type(lexer_t), intent(inout) :: lexer
  type(stream_t), intent(in), target :: stream
  lexer%stream => stream
end subroutine lexer_assign_stream
```

The lexer. The lexer function takes the lexer and returns the currently stored lexeme. If there is none, it is read from buffer, matching against the lexeme templates in the lexer setup. Empty lexemes, i.e., comments and whitespace, are discarded and the buffer is read again until we have found a nonempty lexeme (which may also be EOF or an error condition).

The initial state of the lexer contains an empty lexeme, so reading from buffer is forced. The empty state is restored after returning the lexeme. A nonempty lexeme is present in the lexer only if `lex_back` has been executed before.

The workspace is the `lexer%buffer`, treated as a sort of input stream. We chop off lexemes from the beginning, adjusting the buffer to the left. Whenever the buffer is empty, or we are matching against an open quote which has not terminated, we read a new line and append it to the right. This may result in special conditions, which for simplicity are also returned as lexemes: I/O error, buffer overflow, end of file. If the latter happens during reading a quoted string, we return an unmatched-quote lexeme. Obviously, the special-condition lexemes have to be caught by the parser.

Note that reading further lines is only necessary when reading a quoted string. Otherwise, the line-feed that ends each line is interpreted as whitespace which terminates a preceding lexeme, so there are no other valid multiline lexemes.

To enable meaningful error messages, we also keep track of the line number of the last line read, and the beginning and the end of the current lexeme with respect to this line.

The lexer is implemented as a function that returns the next lexeme (i.e., token). It uses the `lexer` setup and modifies the buffers and pointers stored within the lexer, a side effect. The lexer reads from an input stream object, which also is modified by this reading, e.g., a line pointer is advanced.

⟨*Lexer: public*⟩+≡
```
    public :: lex
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine lex (lexeme, lexer)
      type(lexeme_t), intent(out) :: lexeme
      type(lexer_t), intent(inout) :: lexer
      integer :: iostat1, iostat2
      integer :: pos
      integer, dimension(2) :: range
      integer :: template_index, type
      if (.not. associated (lexer%stream)) &
          call msg_bug ("Lexer called without assigned stream")
      GET_LEXEME: do while (lexeme_get_type (lexer%lexeme) == EMPTY)
        if (len (lexer%buffer) /= 0) then
          iostat1 = 0
        else
          call lexer_read_line (lexer, iostat1)
        end if
        select case (iostat1)
        case (0)
          MATCH_BUFFER: do
            call match (lexer%setup%tt, char (lexer%buffer), &
                        pos, range, template_index)
            if (pos >= 0) then
              type = lexer%setup%type(template_index)
              exit MATCH_BUFFER
            else
              pos = 0
              call lexer_read_line (lexer, iostat2)
              select case (iostat2)
              case (EOF); type = UNMATCHED_QUOTE; exit MATCH_BUFFER
              case (1);   type = IO_ERROR;        exit MATCH_BUFFER
              case (2);   type = OVERFLOW;        exit MATCH_BUFFER
```

```
            end select
          end if
        end do MATCH_BUFFER
      case (EOF); type = EOF
      case (1);   type = IO_ERROR
      case (2);   type = OVERFLOW
      end select
      call lexeme_set (lexer%lexeme, lexer%setup%keyword_list, &
          extract (lexer%buffer, finish=pos), range, type, &
          lexer%setup%keyword_case)
      lexer%buffer = remove (lexer%buffer, finish=pos)
      lexer%previous_column = lexer%current_column
      lexer%current_column = lexer%current_column + pos
    end do GET_LEXEME
    lexeme = lexer%lexeme
    call lexeme_clear (lexer%lexeme)
  end subroutine lex
```

Read a line and append it to the input buffer. If the input buffer overflows, return `iostat=2`. Otherwise, `iostat=1` indicates an I/O error, and `iostat=-1` the EOF.

The input stream may either be an external unit or a `ifile` object. In the latter case, a line is read and the line pointer is advanced.

Note that inserting `LF` between input lines is the Unix convention. Since we are doing this explicitly when gluing lines together, we can pattern-match against `LF` without having to worry about the system.

⟨*Lexer: procedures*⟩+≡
```
  subroutine lexer_read_line (lexer, iostat)
    type(lexer_t), intent(inout) :: lexer
    integer, intent(out) :: iostat
    type(string_t) :: current_line
    current_line = lexer%current_line
    call stream_get_record (lexer%stream, lexer%current_line, iostat)
    if (iostat == 0) then
      lexer%lines_read = lexer%lines_read + 1
      lexer%previous_line2 = lexer%previous_line1
      lexer%previous_line1 = current_line
      lexer%buffer = lexer%buffer // lexer%current_line // LF
      lexer%previous_column = 0
      lexer%current_column = 0
    end if
  end subroutine lexer_read_line
```

Once in a while we have read one lexeme to many, which can be pushed back into the input stream. Do not do this more than once.

⟨*Lexer: public*⟩+≡
```
  public :: lexer_put_back
```

⟨*Lexer: procedures*⟩+≡
```
  subroutine lexer_put_back (lexer, lexeme)
    type(lexer_t), intent(inout) :: lexer
    type(lexeme_t), intent(in) :: lexeme
    if (lexeme_get_type (lexer%lexeme) == EMPTY) then
```

```
            lexer%lexeme = lexeme
        else
            call msg_bug (" Lexer: lex_back fails; probably called twice")
        end if
    end subroutine lexer_put_back
```

3.2.8   Diagnostics

For debugging: print just the setup

⟨*Lexer: public*⟩+≡
```
    public :: lexer_write_setup
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexer_write_setup (lexer, unit)
      type(lexer_t), intent(in) :: lexer
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call lexer_setup_write (lexer%setup, u)
    end subroutine lexer_write_setup
```

This is useful for error printing: show the current line with index and a pointer
to the current column within the line.

⟨*Lexer: public*⟩+≡
```
    public :: lexer_show_location
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexer_show_location (lexer)
      type(lexer_t), intent(in) :: lexer
      type(string_t) :: loc_str
      if (associated (lexer%parent)) then
         call lexer_show_source (lexer%parent)
         call msg_message ("[includes]")
      else
         call msg_message ()
      end if
      if (associated (lexer%stream)) then
         call msg_message &
              (char (stream_get_record_info_string (lexer%stream)) // ":")
      end if
      if (lexer%lines_read >= 4)  call msg_result ("[...]")
      if (lexer%lines_read >= 3)  call msg_result (char (lexer%previous_line2))
      if (lexer%lines_read >= 2)  call msg_result (char (lexer%previous_line1))
      if (lexer%lines_read >= 1) then
         call msg_result (char (lexer%current_line))
         loc_str = repeat (" ", lexer%previous_column)
         loc_str = loc_str // "^"
         if (lexer%current_column > lexer%previous_column) then
            loc_str = loc_str &
                 // repeat ("-", max (lexer%current_column &
                                      - lexer%previous_column - 1, 0)) &
                 // "^"
         end if
```

```
          call msg_result (char (loc_str))
       end if
    end subroutine lexer_show_location
```

This just prints the current stream source.

⟨*Lexer: procedures*⟩+≡
```
    recursive subroutine lexer_show_source (lexer)
      type(lexer_t), intent(in) :: lexer
      type(string_t) :: loc_str
      if (associated (lexer%parent)) then
         call lexer_show_source (lexer%parent)
         call msg_message ("[includes]")
      else
         call msg_message ()
      end if
      if (associated (lexer%stream)) then
         call msg_message &
              (char (stream_get_source_info_string (lexer%stream)) // ":")
      end if
    end subroutine lexer_show_source
```

Test the lexer by lexing and printing all lexemes from unit u, one per line, using preset conventions

⟨*Lexer: public*⟩+≡
```
    public :: lexer_test
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexer_test (lexer, unit)
      type(lexer_t), intent(inout) :: lexer
      integer, intent(in) :: unit
      type(stream_t), target :: stream
      type(lexeme_t) :: lexeme
      call lexer_clear (lexer)
      call stream_init (stream, unit)
      call lexer_assign_stream (lexer, stream)
      do
         call lex (lexeme, lexer)
         call lexeme_write (lexeme, 6)
         if (lexeme_is_break (lexeme))  exit
      end do
      call stream_final (stream)
    end subroutine lexer_test
```

3.3 Syntax rules

This module provides tools to handle syntax rules in an abstract way.

⟨`syntax_rules.f90`⟩≡
  ⟨*File header*⟩

```
module syntax_rules
```

  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
  use limits, only: UNQUOTED !NODEP!
  use diagnostics !NODEP!
  use ifiles, only: line_p, line_init, line_get_string_advance, line_final
  use ifiles, only: ifile_t, ifile_get_length
  use lexers
```

  ⟨*Standard module head*⟩

  ⟨*Syntax: public*⟩

  ⟨*Syntax: parameters*⟩

  ⟨*Syntax: types*⟩

  ⟨*Syntax: interfaces*⟩

```
contains
```

  ⟨*Syntax: subroutines*⟩

```
end module syntax_rules
```

3.3.1 Syntax rules

Syntax rules are used by the parser. They determine how to translate the stream of lexemes as returned by the lexer into the parse tree node. A rule may be terminal, i.e., replace a matching lexeme into a terminal node. The node will contain the lexeme interpreted as a recognized token:

- a keyword: unquoted fixed character string;

- a real number, to be determined at runtime;

- an integer, to be determined at runtime;

- a boolean value, to be determined at runtime;

- a quoted token (e.g., string), to be determined at runtime;

- an identifier (unquoted string that is not a recognized keyword), to be determined at runtime.

It may be nonterminal, i.e., contain a sequence of child rules. These are matched consecutively (and recursively) against the input stream; the resulting node will be a branch node.

- the file, i.e., the input stream as a whole;

- a sequence of syntax elements, where the last syntax element may be optional, or optional repetitive;

Sequences carry a flag that tells whether the last child is optional or may be repeated an arbitrary number of times, correponding to the regexp modifiers `?`, `*`, and `+`.

We also need an alternative rule; this will be replaced by the node generated by one of its children that matches; thus, it does not create a node of its own.

- an alternative of syntax elements.

We also define special types of sequences as convenience macros:

- a list: a sequence where the elements are separated by a separator keyword (e.g., commas), the separators are thrown away when parsing the list;

- a group: a sequence of three tokens, where the first and third ones are left and right delimiters, the delimiters are thrown away;

- an argument list: a delimited list, containing both delimiters and separators.

It would be great to have a polymorphic type for this purpose, but until Fortran 2003 is out we have to emulate this.

Here are the syntax element codes:

⟨*Syntax: public*⟩≡
```
public :: S_UNKNOWN
public :: S_LOGICAL, S_INTEGER, S_REAL, S_COMPLEX, S_QUOTED
public :: S_IDENTIFIER, S_KEYWORD
public :: S_SEQUENCE, S_LIST, S_GROUP, S_ARGS
public :: S_ALTERNATIVE
public :: S_IGNORE
```

⟨*Syntax: parameters*⟩≡
```
integer, parameter :: &
    S_UNKNOWN = 0, &
    S_LOGICAL = 1, S_INTEGER = 2, S_REAL = 3, S_COMPLEX = 4, &
    S_QUOTED = 5, S_IDENTIFIER = 6, S_KEYWORD = 7, &
    S_SEQUENCE = 8, S_LIST = 9, S_GROUP = 10, S_ARGS = 11, &
    S_ALTERNATIVE = 12, &
    S_IGNORE = 99
```

We need arrays of rule pointers, therefore this construct.

⟨*Syntax: types*⟩≡
```
type :: rule_p
   private
   type(syntax_rule_t), pointer :: p => null ()
end type rule_p
```

Return the association status of the rule pointer:

⟨*Syntax: subroutines*⟩≡
```
    elemental function rule_is_associated (rp) result (ok)
      logical :: ok
      type (rule_p), intent(in) :: rp
      ok = associated (rp%p)
    end function rule_is_associated
```

The rule type is one of the types listed above, represented by an integer code. The keyword, for a non-keyword rule, is an identifier used for the printed syntax table. The array of children is needed for nonterminal rules. In that case, there is a modifier for the last element (blank, "?", "*", or "+"), mirrored in the flags `opt` and `rep`. Then, we have the character constants used as separators and delimiters for this rule. Finally, the `used` flag can be set to indicate that this rule is the child of another rule.

⟨*Syntax: types*⟩+≡
```
    public :: syntax_rule_t
```

⟨*Syntax: types*⟩+≡
```
    type :: syntax_rule_t
      private
      integer :: type = S_UNKNOWN
      logical :: used = .false.
      type(string_t) :: keyword
      type(string_t) :: separator
      type(string_t), dimension(2) :: delimiter
      type(rule_p), dimension(:), allocatable :: child
      character(1) :: modifier = ""
      logical :: opt = .false., rep = .false.
    end type syntax_rule_t
```

Initializer: Set type and key for a rule, but do not (yet) allocate anything. Finalizer: not needed (no pointer components).

⟨*Syntax: subroutines*⟩+≡
```
    subroutine syntax_rule_init (rule, key, type)
      type(syntax_rule_t), intent(inout) :: rule
      type(string_t), intent(in) :: key
      integer, intent(in) :: type
      rule%keyword = key
      rule%type = type
      select case (rule%type)
      case (S_GROUP)
         call syntax_rule_set_delimiter (rule)
      case (S_LIST)
         call syntax_rule_set_separator (rule)
      case (S_ARGS)
         call syntax_rule_set_delimiter (rule)
         call syntax_rule_set_separator (rule)
      end select
    end subroutine syntax_rule_init
```

3.3.2 I/O

These characters will not be enclosed in quotes when writing syntax rules:

⟨Limits: public parameters⟩+≡
```
    character(*), parameter, public :: &
            UNQUOTED = "(),|_"//LCLETTERS//UCLETTERS//DIGITS
```

Write an account of the rule. Setting `short` true will suppress the node type. Setting `key_only` true will suppress the definition. Setting `advance` false will suppress the trailing newline.

⟨Syntax: public⟩+≡
```
    public :: syntax_rule_write
```

⟨Syntax: subroutines⟩+≡
```
    subroutine syntax_rule_write (rule, unit, short, key_only, advance)
      type(syntax_rule_t), intent(in) :: rule
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: short, key_only, advance
      logical :: typ, def, adv
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      typ = .true.;  if (present (short))    typ = .not. short
      def = .true.;  if (present (key_only)) def = .not. key_only
      adv = .true.;  if (present (advance))  adv = advance
      select case (rule%type)
      case (S_UNKNOWN);    call write_atom ("???", typ)
      case (S_IGNORE);     call write_atom ("IGNORE", typ)
      case (S_LOGICAL);    call write_atom ("LOGICAL", typ)
      case (S_INTEGER);    call write_atom ("INTEGER", typ)
      case (S_REAL);       call write_atom ("REAL", typ)
      case (S_COMPLEX);    call write_atom ("COMPLEX", typ)
      case (S_IDENTIFIER); call write_atom ("IDENTIFIER", typ)
      case (S_KEYWORD);    call write_atom ("KEYWORD", typ)
      case (S_QUOTED)
         call write_quotes (typ, def, del = rule%delimiter)
      case (S_SEQUENCE)
         call write_sequence ("SEQUENCE", typ, def, size (rule%child))
      case (S_GROUP)
         call write_sequence ("GROUP", typ, def, size (rule%child), &
              del = rule%delimiter)
      case (S_LIST)
         call write_sequence ("LIST", typ, def, size (rule%child), &
              sep = rule%separator)
      case (S_ARGS)
         call write_sequence ("ARGUMENTS", typ, def, size (rule%child), &
              del = rule%delimiter, sep = rule%separator)
      case (S_ALTERNATIVE)
         call write_sequence ("ALTERNATIVE", typ, def, size (rule%child), &
              sep = var_str ("|"))
      end select
      if (adv)  write (u, *)
    contains
      subroutine write_type (type)
        character(*), intent(in) :: type
        character(11) :: str
```

```fortran
      str = type
      write (u, "(1x,A)", advance="no")  str
    end subroutine write_type
    subroutine write_key
      write (u, "(1x,A)", advance="no")  char (wkey (rule))
    end subroutine write_key
    subroutine write_atom (type, typ)
      character(*), intent(in) :: type
      logical, intent(in) :: typ
      if (typ)  call write_type (type)
      call write_key
    end subroutine write_atom
    subroutine write_maybe_quoted (string)
      character(*), intent(in) :: string
      character, parameter :: q = "'"
      character, parameter :: qq = '"'
      if (verify (string, UNQUOTED) == 0) then
         write (u, "(1x,A)", advance = "no") trim (string)
      else if (verify (string, q) == 0) then
         write (u, "(1x,A)", advance = "no") qq // trim (string) // qq
      else
         write (u, "(1x,A)", advance = "no") q // trim (string) // q
      end if
    end subroutine write_maybe_quoted
    subroutine write_quotes (typ, def, del)
      logical, intent(in) :: typ, def
      type(string_t), dimension(2), intent(in) :: del
      if (typ)  call write_type ("QUOTED")
      call write_key
      if (def) then
         write (u, "(1x,'=')", advance="no")
         call write_maybe_quoted (char (del(1)))
         write (u, "(1x,A)", advance="no") "..."
         call write_maybe_quoted (char (del(2)))
      end if
    end subroutine write_quotes
    subroutine write_sequence (type, typ, def, n, del, sep)
      character(*), intent(in) :: type
      logical, intent(in) :: typ, def
      integer, intent(in) :: n
      type(string_t), dimension(2), intent(in), optional :: del
      type(string_t), intent(in), optional :: sep
      integer :: i
      if (typ)  call write_type (type)
      call write_key
      if (def) then
         write (u, "(1x,'=')", advance="no")
         if (present (del))  call write_maybe_quoted (char (del(1)))
         do i = 1, n
            if (i > 1 .and. present (sep)) &
                 call write_maybe_quoted (char (sep))
            write (u, "(1x,A)", advance="no")  &
                 char (wkey (syntax_rule_get_sub_ptr(rule, i)))
            if (i == n)  write (u, "(A)", advance="no")  trim (rule%modifier)
```

```
         end do
         if (present (del))  call write_maybe_quoted (char (del(2)))
       end if
     end subroutine write_sequence
   end subroutine syntax_rule_write
```

In the printed representation, the keyword strings are enclosed as `<...>`, unless
they are bare keywords. Bare keywords are enclosed as `'..'` if they contain a
character which is not a letter, digit, or underscore. If they contain a single-
quote character, they are enclosed as `".."`. (A keyword must not contain both
single- and double-quotes.)

⟨*Syntax: subroutines*⟩+≡
```
   function wkey (rule) result (string)
     type(string_t) :: string
     type(syntax_rule_t), intent(in) :: rule
     select case (rule%type)
     case (S_KEYWORD)
        if (verify (rule%keyword, UNQUOTED) == 0) then
           string = rule%keyword
        else if (scan (rule%keyword, "'") == 0) then
           string = "'" // rule%keyword // "'"
        else
           string = '"' // rule%keyword // '"'
        end if
     case default
        string = "<" // rule%keyword // ">"
     end select
   end function wkey
```

3.3.3 Completing syntax rules

Set the separator and delimiter entries, using defaults:

⟨*Syntax: subroutines*⟩+≡
```
   subroutine syntax_rule_set_separator (rule, separator)
     type(syntax_rule_t), intent(inout) :: rule
     type(string_t), intent(in), optional :: separator
     if (present (separator)) then
        rule%separator = separator
     else
        rule%separator = ","
     end if
   end subroutine syntax_rule_set_separator

   subroutine syntax_rule_set_delimiter (rule, delimiter)
     type(syntax_rule_t), intent(inout) :: rule
     type(string_t), dimension(2), intent(in), optional :: delimiter
     if (present (delimiter)) then
        rule%delimiter = delimiter
     else
        rule%delimiter = (/ "(", ")" /)
     end if
```

```
      end subroutine syntax_rule_set_delimiter
```

Set the modifier entry and corresponding flags:

⟨*Syntax: subroutines*⟩+≡
```
    function is_modifier (string) result (ok)
      logical :: ok
      type(string_t), intent(in) :: string
      select case (char (string))
      case (" ", "?", "*", "+");  ok = .true.
      case default;               ok = .false.
      end select
    end function is_modifier

    subroutine syntax_rule_set_modifier (rule, modifier)
      type(syntax_rule_t), intent(inout) :: rule
      type(string_t), intent(in) :: modifier
      rule%modifier = char (modifier)
      select case (rule%modifier)
      case (" ")
      case ("?");  rule%opt = .true.
      case ("*");  rule%opt = .true.;  rule%rep = .true.
      case ("+");  rule%rep = .true.
      case default
         call msg_bug (" Syntax: sequence modifier '" // rule%modifier &
               // "' is not one of '+' '*' '?'")
      end select
    end subroutine syntax_rule_set_modifier
```

Check a finalized rule for completeness

⟨*Syntax: subroutines*⟩+≡
```
    subroutine syntax_rule_check (rule)
      type(syntax_rule_t), intent(in) :: rule
      if (rule%keyword == "")  call msg_bug ("Rule key not set")
      select case (rule%type)
      case (S_UNKNOWN);  call bug (" Undefined rule")
      case (S_IGNORE, S_LOGICAL, S_INTEGER, S_REAL, S_COMPLEX, &
              S_IDENTIFIER, S_KEYWORD)
      case (S_QUOTED)
         if (any (rule%delimiter == "")) call bug (" Missing quote character(s)")
      case (S_SEQUENCE)
      case (S_GROUP)
         if (any (rule%delimiter == "")) call bug (" Missing delimiter(s)")
      case (S_LIST)
         if (rule%separator == "") call bug (" Missing separator")
      case (S_ARGS)
         if (any (rule%delimiter == "")) call bug (" Missing delimiter(s)")
         if (rule%separator == "") call bug (" Missing separator")
      case (S_ALTERNATIVE)
      case default
         call bug (" Undefined syntax code")
      end select
      select case (rule%type)
      case (S_SEQUENCE, S_GROUP, S_LIST, S_ARGS, S_ALTERNATIVE)
```

```
            if (allocated (rule%child)) then
                if (.not.all (rule_is_associated (rule%child))) &
                    call bug (" Child rules not all associated")
            else
                call bug (" Parent rule without children")
            end if
        case default
            if (allocated (rule%child))  call bug (" Non-parent rule with children")
        end select
    contains
      subroutine bug (string)
        character(*), intent(in) :: string
        call msg_bug (" Syntax table: Rule " // char (rule%keyword) // ": " &
            // string)
      end subroutine bug
    end subroutine syntax_rule_check
```

3.3.4 Accessing rules

This is the API for syntax rules:

⟨*Syntax: public*⟩+≡
```
    public :: syntax_rule_get_type
```

⟨*Syntax: subroutines*⟩+≡
```
    function syntax_rule_get_type (rule) result (type)
      integer :: type
      type(syntax_rule_t), intent(in) :: rule
      type = rule%type
    end function syntax_rule_get_type
```

⟨*Syntax: public*⟩+≡
```
    public :: syntax_rule_get_key
```

⟨*Syntax: subroutines*⟩+≡
```
    function syntax_rule_get_key (rule) result (key)
      type(string_t) :: key
      type(syntax_rule_t), intent(in) :: rule
      key = rule%keyword
    end function syntax_rule_get_key
```

⟨*Syntax: public*⟩+≡
```
    public :: syntax_rule_get_separator
    public :: syntax_rule_get_delimiter
```

⟨*Syntax: subroutines*⟩+≡
```
    function syntax_rule_get_separator (rule) result (separator)
      type(string_t) :: separator
      type(syntax_rule_t), intent(in) :: rule
      separator = rule%separator
    end function syntax_rule_get_separator

    function syntax_rule_get_delimiter (rule) result (delimiter)
      type(string_t), dimension(2) :: delimiter
```

```
      type(syntax_rule_t), intent(in) :: rule
      delimiter = rule%delimiter
    end function syntax_rule_get_delimiter
```

Accessing child rules. If we use `syntax_rule_get_n_sub` for determining loop bounds, we do not need a check in the second routine.

⟨*Syntax: public*⟩+≡
```
    public :: syntax_rule_get_n_sub
    public :: syntax_rule_get_sub_ptr
```

⟨*Syntax: subroutines*⟩+≡
```
    function syntax_rule_get_n_sub (rule) result (n)
      integer :: n
      type(syntax_rule_t), intent(in) :: rule
      if (allocated (rule%child)) then
         n = size (rule%child)
      else
         n = 0
      end if
    end function syntax_rule_get_n_sub

    function syntax_rule_get_sub_ptr (rule, i) result (sub)
      type(syntax_rule_t), pointer :: sub
      type(syntax_rule_t), intent(in), target :: rule
      integer, intent(in) :: i
      sub => rule%child(i)%p
    end function syntax_rule_get_sub_ptr

    subroutine syntax_rule_set_sub (rule, i, sub)
      type(syntax_rule_t), intent(inout) :: rule
      integer, intent(in) :: i
      type(syntax_rule_t), intent(in), target :: sub
      rule%child(i)%p => sub
    end subroutine syntax_rule_set_sub
```

Return the modifier flags:

⟨*Syntax: public*⟩+≡
```
    public :: syntax_rule_last_optional
    public :: syntax_rule_last_repetitive
```

⟨*Syntax: subroutines*⟩+≡
```
    function syntax_rule_last_optional (rule) result (opt)
      logical :: opt
      type(syntax_rule_t), intent(in) :: rule
      opt = rule%opt
    end function syntax_rule_last_optional
    function syntax_rule_last_repetitive (rule) result (rep)
      logical :: rep
      type(syntax_rule_t), intent(in) :: rule
      rep = rule%rep
    end function syntax_rule_last_repetitive
```

Return true if the rule is atomic, i.e., logical, real, keyword etc.

⟨*Syntax: public*⟩+≡
```
    public :: syntax_rule_is_atomic
```

⟨*Syntax: subroutines*⟩+≡
```
    function syntax_rule_is_atomic (rule) result (atomic)
      logical :: atomic
      type(syntax_rule_t), intent(in) :: rule
      select case (rule%type)
      case (S_LOGICAL, S_INTEGER, S_REAL, S_COMPLEX, S_IDENTIFIER, &
              S_KEYWORD, S_QUOTED)
         atomic = .true.
      case default
         atomic = .false.
      end select
    end function syntax_rule_is_atomic
```

3.3.5   Syntax tables

A syntax table contains the tree of syntax rules and, for direct parser access, the list of valid keywords.

Types

The syntax contains an array of rules and a list of keywords. The array is actually used as a tree, where the top rule is the first array element, and the other rules are recursively pointed to by this first rule. (No rule should be used twice or be unused.) The keyword list is derived from the rule tree.

Objects of this type need the target attribute if they are associated with a lexer. The keyword list will be pointed to by this lexer.

⟨*Syntax: public*⟩+≡
```
    public :: syntax_t
```

⟨*Syntax: types*⟩+≡
```
    type :: syntax_t
      private
      type(syntax_rule_t), dimension(:), allocatable :: rule
      type(keyword_list_t) :: keyword_list
    end type syntax_t
```

Constructor/destructor

Initialize and finalize syntax tables

⟨*Syntax: public*⟩+≡
```
    public :: syntax_init
    public :: syntax_final
```

There are two ways to create a syntax: hard-coded from rules or dynamically from file.

⟨*Syntax: interfaces*⟩≡
```
interface syntax_init
   module procedure syntax_init_from_ifile
end interface
```

The syntax definition is read from an `ifile` object which contains the syntax definitions in textual form, one rule per line. This interface allows for determining the number of rules beforehand.

To parse the rule definitions, we make up a temporary lexer. Obviously, we cannot use a generic parser yet, so we have to hardcode the parsing process.

⟨*Syntax: subroutines*⟩+≡
```
subroutine syntax_init_from_ifile (syntax, ifile)
  type(syntax_t), intent(out), target :: syntax
  type(ifile_t), intent(in) :: ifile
  type(lexer_t) :: lexer
  type(line_p) :: line
  type(string_t) :: string
  integer :: n_token
  integer :: i
  call lexer_init (lexer, &
     comment_chars = "", &
     quote_chars = "<'""", &
     quote_match = ">'""", &
     single_chars = "?*+|=,()", &
     special_class = (/ "." /), &
     keyword_list = null ())
  allocate (syntax%rule (ifile_get_length (ifile)))
  call line_init (line, ifile)
  do i = 1, size (syntax%rule)
     string = line_get_string_advance (line)
     call set_rule_type_and_key (syntax%rule(i), string, lexer)
  end do
  call line_init (line, ifile)
  do i = 1, size (syntax%rule)
     string = line_get_string_advance (line)
     select case (syntax%rule(i)%type)
     case (S_QUOTED, S_SEQUENCE, S_GROUP, S_LIST, S_ARGS, S_ALTERNATIVE)
        n_token = get_n_token (string, lexer)
        call set_rule_contents &
             (syntax%rule(i), syntax, n_token, string, lexer)
     end select
  end do
  call line_final (line)
  call lexer_final (lexer)
  call syntax_make_keyword_list (syntax)
  if (.not. all (syntax%rule%used)) then
     do i = 1, size (syntax%rule)
        if (.not. syntax%rule(i)%used) then
           call syntax_rule_write (syntax%rule(i), 6)
        end if
     end do
```

```
        call msg_bug (" Syntax table: unused rules")
      end if
    end subroutine syntax_init_from_ifile
```

For a given rule defined in the input, the first task is to determine its type and key. With these, we can initialize the rule in the table, postponing the association of children.

⟨*Syntax: subroutines*⟩+≡
```
    subroutine set_rule_type_and_key (rule, string, lexer)
      type(syntax_rule_t), intent(inout) :: rule
      type(string_t), intent(in) :: string
      type(lexer_t), intent(inout) :: lexer
      type(stream_t), target :: stream
      type(lexeme_t) :: lexeme
      type(string_t) :: key
      character(2) :: type
      call lexer_clear (lexer)
      call stream_init (stream, string)
      call lexer_assign_stream (lexer, stream)
      call lex (lexeme, lexer)
      type = lexeme_get_string (lexeme)
      call lex (lexeme, lexer)
      key = lexeme_get_contents (lexeme)
      call stream_final (stream)
      if (trim (key) /= "") then
         select case (type)
         case ("IG");  call syntax_rule_init (rule, key, S_IGNORE)
         case ("LO");  call syntax_rule_init (rule, key, S_LOGICAL)
         case ("IN");  call syntax_rule_init (rule, key, S_INTEGER)
         case ("RE");  call syntax_rule_init (rule, key, S_REAL)
         case ("CO");  call syntax_rule_init (rule, key, S_COMPLEX)
         case ("ID");  call syntax_rule_init (rule, key, S_IDENTIFIER)
         case ("KE");  call syntax_rule_init (rule, key, S_KEYWORD)
         case ("QU");  call syntax_rule_init (rule, key, S_QUOTED)
         case ("SE");  call syntax_rule_init (rule, key, S_SEQUENCE)
         case ("GR");  call syntax_rule_init (rule, key, S_GROUP)
         case ("LI");  call syntax_rule_init (rule, key, S_LIST)
         case ("AR");  call syntax_rule_init (rule, key, S_ARGS)
         case ("AL");  call syntax_rule_init (rule, key, S_ALTERNATIVE)
         case default
            call lexer_show_location (lexer)
            call msg_bug (" Syntax definition: unknown type '" // type // "'")
         end select
      else
         print *, char (string)
         call msg_bug (" Syntax definition: empty rule key")
      end if
    end subroutine set_rule_type_and_key
```

This function returns the number of tokens in an input line.

⟨*Syntax: subroutines*⟩+≡
```
    function get_n_token (string, lexer) result (n)
      integer :: n
```

```
      type(string_t), intent(in) :: string
      type(lexer_t), intent(inout) :: lexer
      type(stream_t), target :: stream
      type(lexeme_t) :: lexeme
      integer :: i
      call lexer_clear (lexer)
      call stream_init (stream, string)
      call lexer_assign_stream (lexer, stream)
      i = 0
      do
         call lex (lexeme, lexer)
         if (lexeme_is_break (lexeme))  exit
         i = i + 1
      end do
      n = i
      call stream_final (stream)
  end function get_n_token
```

This subroutine extracts the rule contents for an input line. There are three tasks: (1) determine the number of children, depending on the rule type; (2) find and set the separator and delimiter strings, if required; (3) scan the child rules, find them in the syntax table and associate the parent rule with them.

⟨*Syntax: subroutines*⟩+≡

```
    subroutine set_rule_contents (rule, syntax, n_token, string, lexer)
      type(syntax_rule_t), intent(inout) :: rule
      type(syntax_t), intent(in), target :: syntax
      integer, intent(in) :: n_token
      type(string_t), intent(in) :: string
      type(lexer_t), intent(inout) :: lexer
      type(stream_t), target :: stream
      type(lexeme_t), dimension(n_token) :: lexeme
      integer :: i, n_children
      call lexer_clear (lexer)
      call stream_init (stream, string)
      call lexer_assign_stream (lexer, stream)
      do i = 1, n_token
         call lex (lexeme(i), lexer)
      end do
      call stream_final (stream)
      n_children = get_n_children ()
      call set_delimiters
      if (n_children > 1)  call set_separator
      if (n_children > 0)  call set_children
    contains
      function get_n_children () result (n_children)
        integer :: n_children
        select case (rule%type)
        case (S_QUOTED)
           if (n_token /= 6)  call broken_rule (rule)
           n_children = 0
        case (S_GROUP)
           if (n_token /= 6)  call broken_rule (rule)
           n_children = 1
```

```
    case (S_SEQUENCE)
       if (is_modifier (lexeme_get_string (lexeme(n_token)))) then
          if (n_token <= 4)  call broken_rule (rule)
          call syntax_rule_set_modifier &
                (rule, lexeme_get_string (lexeme(n_token)))
          n_children = n_token - 4
       else
          if (n_token <= 3) call broken_rule (rule)
          n_children = n_token - 3
       end if
    case (S_LIST)
       if (is_modifier (lexeme_get_string (lexeme(n_token)))) then
          if (n_token <= 4 .or. mod (n_token, 2) /= 1) &
                call broken_rule (rule)
          call syntax_rule_set_modifier &
                (rule, lexeme_get_string (lexeme(n_token)))
       else if (n_token <= 3 .or. mod (n_token, 2) /= 0) then
          call broken_rule (rule)
       end if
       n_children = (n_token - 2) / 2
    case (S_ARGS)
       if (is_modifier (lexeme_get_string (lexeme(n_token-1)))) then
          if (n_token <= 6 .or. mod (n_token, 2) /= 1) &
                call broken_rule (rule)
          call syntax_rule_set_modifier &
                (rule, lexeme_get_string (lexeme(n_token-1)))
       else if (n_token <= 5 .or. mod (n_token, 2) /= 0) then
          call broken_rule (rule)
       end if
       n_children = (n_token - 4) / 2
    case (S_ALTERNATIVE)
       if (n_token <= 3 .or. mod (n_token, 2) /= 0)  call broken_rule (rule)
       n_children = (n_token - 2) / 2
    end select
  end function get_n_children
  subroutine set_delimiters
    type(string_t), dimension(2) :: delimiter
    select case (rule%type)
    case (S_QUOTED, S_GROUP, S_ARGS)
       delimiter(1) = lexeme_get_contents (lexeme(4))
       delimiter(2) = lexeme_get_contents (lexeme(n_token))
       call syntax_rule_set_delimiter (rule, delimiter)
    end select
  end subroutine set_delimiters
  subroutine set_separator
    type(string_t) :: separator
    select case (rule%type)
    case (S_LIST)
       separator = lexeme_get_contents (lexeme(5))
       call syntax_rule_set_separator (rule, separator)
    case (S_ARGS)
       separator = lexeme_get_contents (lexeme(6))
       call syntax_rule_set_separator (rule, separator)
    end select
```

```
      end subroutine set_separator
    subroutine set_children
      allocate (rule%child(n_children))
      select case (rule%type)
      case (S_GROUP)
         call syntax_rule_set_sub (rule, 1, syntax_get_rule_ptr (syntax, &
              lexeme_get_contents (lexeme(5))))
      case (S_SEQUENCE)
         do i = 1, n_children
            call syntax_rule_set_sub (rule, i, syntax_get_rule_ptr (syntax, &
                 lexeme_get_contents (lexeme(i+3))))
         end do
      case (S_LIST, S_ALTERNATIVE)
         do i = 1, n_children
            call syntax_rule_set_sub (rule, i, syntax_get_rule_ptr (syntax, &
                 lexeme_get_contents (lexeme(2*i+2))))
         end do
      case (S_ARGS)
         do i = 1, n_children
            call syntax_rule_set_sub (rule, i, syntax_get_rule_ptr (syntax, &
                 lexeme_get_contents (lexeme(2*i+3))))
         end do
      end select
    end subroutine set_children
    subroutine broken_rule (rule)
      type(syntax_rule_t), intent(in) :: rule
      call lexer_show_location (lexer)
      call msg_bug (" Syntax definition: broken rule '" &
           // char (wkey (rule)) // "'")
    end subroutine broken_rule
  end subroutine set_rule_contents
```

This routine completes the syntax table object. We assume that the rule array
is set up. We associate the top rule with the first entry in the rule array and
build up the keyword list.

The keyword list includes delimiters and separators. Filling it can only be
done after all rules are set. We scan the rule tree. For each keyword that
we find, we try to add it to the keyword list; the pointer to the last element
is carried along with the recursive scanning. Before appending a keyword, we
check whether it is already in the list.

⟨*Syntax: subroutines*⟩+≡
```
  subroutine syntax_make_keyword_list (syntax)
    type(syntax_t), intent(inout), target :: syntax
    type(syntax_rule_t), pointer :: rule
    rule => syntax%rule(1)
    call rule_scan_rec (rule, syntax%keyword_list)
  contains
    recursive subroutine rule_scan_rec (rule, keyword_list)
      type(syntax_rule_t), pointer :: rule
      type(keyword_list_t), intent(inout) :: keyword_list
      integer :: i
      if (rule%used)  return
      rule%used = .true.
```

```
      select case (rule%type)
      case (S_UNKNOWN)
         call msg_bug (" Syntax: rule tree contains undefined rule")
      case (S_KEYWORD)
         call keyword_list_add (keyword_list, rule%keyword)
      end select
      select case (rule%type)
      case (S_LIST, S_ARGS)
         call keyword_list_add (keyword_list, rule%separator)
      end select
      select case (rule%type)
      case (S_GROUP, S_ARGS)
         call keyword_list_add (keyword_list, rule%delimiter(1))
         call keyword_list_add (keyword_list, rule%delimiter(2))
      end select
      select case (rule%type)
      case (S_SEQUENCE, S_GROUP, S_LIST, S_ARGS, S_ALTERNATIVE)
         if (.not. allocated (rule%child)) &
              call msg_bug (" Syntax: Non-terminal rule without children")
      case default
         if (allocated (rule%child)) &
              call msg_bug (" Syntax: Terminal rule with children")
      end select
      if (allocated (rule%child)) then
         do i = 1, size (rule%child)
            call rule_scan_rec (rule%child(i)%p, keyword_list)
         end do
      end if
    end subroutine rule_scan_rec
  end subroutine syntax_make_keyword_list
```

The finalizer deallocates the rule pointer array and deletes the keyword list.

⟨*Syntax: subroutines*⟩+≡

```
  subroutine syntax_final (syntax)
    type(syntax_t), intent(inout) :: syntax
    if (allocated (syntax%rule))  deallocate (syntax%rule)
    call keyword_list_final (syntax%keyword_list)
  end subroutine syntax_final
```

3.3.6 Accessing the syntax table

Return a pointer to the top rule

⟨*Syntax: public*⟩+≡

```
  public :: syntax_get_top_rule_ptr
```

⟨*Syntax: subroutines*⟩+≡

```
  function syntax_get_top_rule_ptr (syntax) result (rule)
    type(syntax_rule_t), pointer :: rule
    type(syntax_t), intent(in), target :: syntax
    if (allocated (syntax%rule)) then
       rule => syntax%rule(1)
    else
```

```
            rule => null ()
          end if
        end function syntax_get_top_rule_ptr
```

Assign the pointer to the rule associated with a given key (assumes that the rule array is allocated)

⟨*Syntax: public*⟩+≡
```
    public :: syntax_get_rule_ptr
```

⟨*Syntax: subroutines*⟩+≡
```
    function syntax_get_rule_ptr (syntax, key) result (rule)
      type(syntax_rule_t), pointer :: rule
      type(syntax_t), intent(in), target :: syntax
      type(string_t), intent(in) :: key
      integer :: i
      do i = 1, size (syntax%rule)
         if (syntax%rule(i)%keyword == key) then
            rule => syntax%rule(i)
            return
         end if
      end do
      call msg_bug (" Syntax table: Rule " // char (key) // " not found")
    end function syntax_get_rule_ptr
```

Return a pointer to the keyword list

⟨*Syntax: public*⟩+≡
```
    public :: syntax_get_keyword_list_ptr
```

⟨*Syntax: subroutines*⟩+≡
```
    function syntax_get_keyword_list_ptr (syntax) result (keyword_list)
      type(keyword_list_t), pointer :: keyword_list
      type(syntax_t), intent(in), target :: syntax
      keyword_list => syntax%keyword_list
    end function syntax_get_keyword_list_ptr
```

3.3.7 I/O

Write a readable representation of the syntax table

⟨*Syntax: public*⟩+≡
```
    public :: syntax_write
```

⟨*Syntax: subroutines*⟩+≡
```
    subroutine syntax_write (syntax, unit)
      type(syntax_t), intent(in) :: syntax
      integer, intent(in), optional :: unit
      integer :: u
      integer :: i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)") "Syntax table:"
      if (allocated (syntax%rule)) then
         do i = 1, size (syntax%rule)
            call syntax_rule_write (syntax%rule(i), u)
```

```
      end do
  else
      write (u, "(1x,A)") "[not allocated]"
  end if
  call keyword_list_write (syntax%keyword_list, u)
end subroutine syntax_write
```

3.4 The parser

On a small scale, the parser interprets the string tokens returned by the lexer; they are interpreted as numbers, keywords and such and stored as a typed object. On a large scale, a text is read, parsed, and a syntax rule set is applied such that the tokens are stored as a parse tree. Syntax errors are spotted in this process, so the resulting parse tree is syntactically correct by definition.

⟨parser.f90⟩≡
  ⟨*File header*⟩

```
  module parser
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
```
    use limits, only: DIGITS !NODEP!
```
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use md5
    use lexers
    use syntax_rules
```

  ⟨*Standard module head*⟩

  ⟨*Parser: public*⟩

  ⟨*Parser: types*⟩

  ⟨*Parser: interfaces*⟩

```
  contains
```

  ⟨*Parser: procedures*⟩

```
  end module parser
```

3.4.1 The token type

Tokens are elements of the parsed input that carry a value: logical, integer, real, quoted string, (unquoted) identifier, or known keyword. Note that non-keyword tokens also have an abstract key attached to them.

   This is an obvious candidate for polymorphism.

⟨*Parser: types*⟩≡
```
    type :: token_t
       private
       integer :: type = S_UNKNOWN
       logical, pointer :: lval => null ()
       integer, pointer :: ival => null ()
       real(default), pointer :: rval => null ()
       complex(default), pointer :: cval => null ()
       type(string_t), pointer :: sval => null ()
       type(string_t), pointer :: kval => null ()
       type(string_t), dimension(:), pointer :: quote => null ()
```

```
      end type token_t
```

Create a token from the lexeme returned by the lexer: Allocate storage and try to interpret the lexeme according to the type that is requested by the parser. For a keyword token, match the lexeme against the requested key. If successful, set the token type, value, and key. Otherwise, set the type to S_UNKNOWN.

⟨*Parser: procedures*⟩≡

```
    subroutine token_init (token, lexeme, requested_type, key)
      type(token_t), intent(out) :: token
      type(lexeme_t), intent(in) :: lexeme
      integer, intent(in) :: requested_type
      type(string_t), intent(in) :: key
      integer :: type
      type = lexeme_get_type (lexeme)
      token%type = S_UNKNOWN
      select case (requested_type)
      case (S_LOGICAL)
         if (type == T_IDENTIFIER)  call read_logical &
              (char (lexeme_get_string (lexeme)))
      case (S_INTEGER)
         if (type == T_NUMERIC)  call read_integer &
              (char (lexeme_get_string (lexeme)))
      case (S_REAL)
         if (type == T_NUMERIC)  call read_real &
              (char (lexeme_get_string (lexeme)))
      case (S_COMPLEX)
         if (type == T_NUMERIC)  call read_complex &
              (char (lexeme_get_string (lexeme)))
      case (S_IDENTIFIER)
         if (type == T_IDENTIFIER)  call read_identifier &
              (lexeme_get_string (lexeme))
      case (S_KEYWORD)
         if (type == T_KEYWORD)  call check_keyword &
              (lexeme_get_string (lexeme), key)
      case (S_QUOTED)
         if (type == T_QUOTED)   call read_quoted &
              (lexeme_get_contents (lexeme), lexeme_get_delimiters (lexeme))
      case default
         print *, requested_type
         call msg_bug (" Invalid token type code requested by the parser")
      end select
      if (token%type /= S_UNKNOWN) then
         allocate (token%kval)
         token%kval = key
      end if
    contains
      subroutine read_logical (s)
        character(*), intent(in) :: s
        select case (s)
        case ("t", "T", "true", "TRUE", "y", "Y", "yes", "YES")
           allocate (token%lval)
           token%lval = .true.
           token%type = S_LOGICAL
```

```fortran
    case ("f", "F", "false", "FALSE", "n", "N", "no", "NO")
       allocate (token%lval)
       token%lval = .false.
       token%type = S_LOGICAL
    end select
end subroutine read_logical
subroutine read_integer (s)
  character(*), intent(in) :: s
  integer :: tmp, iostat
  if (verify (s, DIGITS) == 0) then
     read (s, *, iostat=iostat) tmp
     if (iostat == 0) then
        allocate (token%ival)
        token%ival = tmp
        token%type = S_INTEGER
     end if
  end if
end subroutine read_integer
subroutine read_real (s)
  character(*), intent(in) :: s
  real(default) :: tmp
  integer :: iostat
  read (s, *, iostat=iostat) tmp
  if (iostat == 0) then
     allocate (token%rval)
     token%rval = tmp
     token%type = S_REAL
  end if
end subroutine read_real
subroutine read_complex (s)
  character(*), intent(in) :: s
  complex(default) :: tmp
  integer :: iostat
  read (s, *, iostat=iostat) tmp
  if (iostat == 0) then
     allocate (token%cval)
     token%cval = tmp
     token%type = S_COMPLEX
  end if
end subroutine read_complex
subroutine read_identifier (s)
  type(string_t), intent(in) :: s
  allocate (token%sval)
  token%sval = s
  token%type = S_IDENTIFIER
end subroutine read_identifier
subroutine check_keyword (s, key)
  type(string_t), intent(in) :: s
  type(string_t), intent(in) :: key
  if (key == s)  token%type = S_KEYWORD
end subroutine check_keyword
subroutine read_quoted (s, del)
  type(string_t), intent(in) :: s
  type(string_t), dimension(2), intent(in) :: del
```

```
      allocate (token%sval, token%quote(2))
      token%sval = s
      token%quote(1) = del(1)
      token%quote(2) = del(2)
      token%type = S_QUOTED
    end subroutine read_quoted
  end subroutine token_init
```

Reset a token to an empty state, freeing allocated memory, and deallocate the token itself.

⟨*Parser: procedures*⟩+≡
```
  subroutine token_final (token)
    type(token_t), intent(inout) :: token
    token%type = S_UNKNOWN
    if (associated (token%lval))  deallocate (token%lval)
    if (associated (token%ival))  deallocate (token%ival)
    if (associated (token%rval))  deallocate (token%rval)
    if (associated (token%sval))  deallocate (token%sval)
    if (associated (token%kval))  deallocate (token%kval)
    if (associated (token%quote))  deallocate (token%quote)
  end subroutine token_final
```

Check for empty=valid token:

⟨*Parser: procedures*⟩+≡
```
  function token_is_valid (token) result (valid)
    logical :: valid
    type(token_t), intent(in) :: token
    valid = token%type /= S_UNKNOWN
  end function token_is_valid
```

Write the contents of a token.

⟨*Parser: procedures*⟩+≡
```
  subroutine token_write (token, unit)
    type(token_t), intent(in) :: token
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit);  if (u < 0)  return
    select case (token%type)
    case (S_LOGICAL)
      write (u, *) token%lval
    case (S_INTEGER)
      write (u, *) token%ival
    case (S_REAL)
      write (u, *) token%rval
    case (S_COMPLEX)
      write (u, *) token%cval
    case (S_IDENTIFIER)
      write (u, *) char (token%sval)
    case (S_KEYWORD)
      write (u, *) '[keyword] ' // char (token%kval)
    case (S_QUOTED)
      write (u, *) &
```

```
            char (token%quote(1)) // char (token%sval) // char (token%quote(2))
      case default
         write (u, *) '[empty]'
      end select
   end subroutine token_write
```

Token assignment via deep copy. This is useful to avoid confusion when the token is transferred to some parse-tree node.

⟨*Parser: interfaces*⟩≡
```
   interface assignment(=)
      module procedure token_assign
   end interface
```

We need to copy only the contents that are actually assigned, the other pointers remain disassociated.

⟨*Parser: procedures*⟩+≡
```
   subroutine token_assign (token, token_in)
     type(token_t), intent(out) :: token
     type(token_t), intent(in) :: token_in
     token%type = token_in%type
     select case (token%type)
     case (S_LOGICAL);    allocate (token%lval);  token%lval = token_in%lval
     case (S_INTEGER);    allocate (token%ival);  token%ival = token_in%ival
     case (S_REAL);       allocate (token%rval);  token%rval = token_in%rval
     case (S_COMPLEX); allocate (token%cval);  token%cval = token_in%cval
     case (S_IDENTIFIER); allocate (token%sval);  token%sval = token_in%sval
     case (S_QUOTED);     allocate (token%sval);  token%sval = token_in%sval
        allocate (token%quote(2));  token%quote = token_in%quote
     end select
     if (token%type /= S_UNKNOWN) then
        allocate (token%kval);  token%kval = token_in%kval
     end if
   end subroutine token_assign
```

3.4.2   Retrieve token contents

These functions all do a trivial sanity check that should avoid crashes.

⟨*Parser: procedures*⟩+≡
```
   function token_get_logical (token) result (lval)
     logical :: lval
     type(token_t), intent(in) :: token
     if (associated (token%lval)) then
        lval = token%lval
     else
        call token_mismatch (token, "logical")
     end if
   end function token_get_logical

   function token_get_integer (token) result (ival)
     integer :: ival
     type(token_t), intent(in) :: token
```

```
      if (associated (token%ival)) then
         ival = token%ival
      else
         call token_mismatch (token, "integer")
      end if
end function token_get_integer

function token_get_real (token) result (rval)
  real(default) :: rval
  type(token_t), intent(in) :: token
  if (associated (token%rval)) then
     rval = token%rval
  else
     call token_mismatch (token, "real")
  end if
end function token_get_real

function token_get_cmplx (token) result (cval)
  real(default) :: cval
  type(token_t), intent(in) :: token
  if (associated (token%cval)) then
     cval = token%cval
  else
     call token_mismatch (token, "complex")
  end if
end function token_get_cmplx

function token_get_string (token) result (sval)
  type(string_t) :: sval
  type(token_t), intent(in) :: token
  if (associated (token%sval)) then
     sval = token%sval
  else
     call token_mismatch (token, "string")
  end if
end function token_get_string

function token_get_key (token) result (kval)
  type(string_t) :: kval
  type(token_t), intent(in) :: token
  if (associated (token%kval)) then
     kval = token%kval
  else
     call token_mismatch (token, "keyword")
  end if
end function token_get_key

function token_get_quote (token) result (quote)
  type(string_t), dimension(2) :: quote
  type(token_t), intent(in) :: token
  if (associated (token%quote)) then
     quote = token%quote
  else
     call token_mismatch (token, "quote")
```

```
        end if
    end function token_get_quote
```

⟨*Parser: procedures*⟩+≡
```
    subroutine token_mismatch (token, type)
      type(token_t), intent(in) :: token
      character(*), intent(in) :: type
      write (6, "(A)", advance="no")  "Token: "
      call token_write (token)
      call msg_bug (" Token type mismatch; value required as " // type)
    end subroutine token_mismatch
```

3.4.3   The parse tree: nodes

The parser will generate a parse tree from the input stream. Each node in this
parse tree points to the syntax rule that was applied. (Since syntax rules are
stored in a pointer-type array within the syntax table, they qualify as targets.)
A leaf node contains a token. A branch node has subnodes. The subnodes are
stored as a list, so each node also has a **next** pointer.

⟨*Parser: public*⟩≡
```
    public :: parse_node_t
```

⟨*Parser: types*⟩+≡
```
    type :: parse_node_t
      private
      type(syntax_rule_t), pointer :: rule => null ()
      type(token_t) :: token
      integer :: n_sub = 0
      type(parse_node_t), pointer :: sub_first => null ()
      type(parse_node_t), pointer :: sub_last => null ()
      type(parse_node_t), pointer :: next => null ()
    end type parse_node_t
```

Output. The first version writes a node together with its sub-node tree, orga-
nized by indentation.

⟨*Parser: public*⟩+≡
```
    public :: parse_node_write_rec
```

⟨*Parser: procedures*⟩+≡
```
    recursive subroutine parse_node_write_rec (node, unit, short, depth)
      type(parse_node_t), intent(in), target :: node
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: short
      integer, intent(in), optional :: depth
      integer :: u, d
      type(parse_node_t), pointer :: current
      u = output_unit (unit);  if (u < 0)  return
      d = 0;  if (present (depth))  d = depth
      call parse_node_write (node, u, short=short)
      current => node%sub_first
      do while (associated (current))
          write (u, "(A)", advance = "no")  repeat ("|  ", d)
```

```
            call parse_node_write_rec (current, unit, short, d+1)
            current => current%next
          end do
       end subroutine parse_node_write_rec
```

This does the actual output for a single node, without recursion.
⟨*Parser: public*⟩+≡
```
    public :: parse_node_write
```

⟨*Parser: procedures*⟩+≡
```
    subroutine parse_node_write (node, unit, short)
      type(parse_node_t), intent(in) :: node
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: short
      integer :: u
      type(parse_node_t), pointer :: current
      u = output_unit (unit);  if (u < 0)  return
      write (u, "('+ ')", advance = "no")
      if (associated (node%rule)) then
         call syntax_rule_write (node%rule, u, &
              short=short, key_only=.true., advance=.false.)
         if (token_is_valid (node%token)) then
            write (u, "('  = ')", advance="no")
            call token_write (node%token, u)
         else if (associated (node%sub_first)) then
            write (u, "('  = ')", advance="no")
            current => node%sub_first
            do while (associated (current))
               call syntax_rule_write (current%rule, u, &
                    short=.true., key_only=.true., advance=.false.)
               current => current%next
            end do
            write (u, *)
         else
            write (u, *)
         end if
      else
         write (u, *) "[empty]"
      end if
    end subroutine parse_node_write
```

Finalize the token and recursively finalize and deallocate all sub-nodes.
⟨*Parser: public*⟩+≡
```
    public :: parse_node_final
```

⟨*Parser: procedures*⟩+≡
```
    recursive subroutine parse_node_final (node, recursive)
      type(parse_node_t), intent(inout) :: node
      type(parse_node_t), pointer :: current
      logical, intent(in), optional :: recursive
      logical :: rec
      rec = .true.;  if (present (recursive))  rec = recursive
      call token_final (node%token)
      if (rec) then
```

```
            do while (associated (node%sub_first))
               current => node%sub_first
               node%sub_first => node%sub_first%next
               call parse_node_final (current)
               deallocate (current)
            end do
         end if
      end subroutine parse_node_final
```

3.4.4 Filling nodes

The constructors allocate and initialize the node. There are two possible initializers (in a later version, should correspond to different type extensions).

First, leaf (terminal) nodes. The token constructor does the actual work, looking at the requested type and key for the given rule and matching against the lexeme contents. If it fails, the token will keep the type S_UNKNOWN and remain empty. Otherwise, we have a valid node which contains the new token.

If the lexeme argument is absent, the token is left empty.

⟨*Parser: procedures*⟩+≡
```
   subroutine parse_node_create_leaf (node, rule, lexeme)
     type(parse_node_t), pointer :: node
     type(syntax_rule_t), intent(in), target :: rule
     type(lexeme_t), intent(in) :: lexeme
     allocate (node)
     node%rule => rule
     call token_init (node%token, lexeme, &
          syntax_rule_get_type (rule), syntax_rule_get_key (rule))
     if (.not. token_is_valid (node%token))  deallocate (node)
   end subroutine parse_node_create_leaf
```

Second, branch nodes. We first assign the rule:

⟨*Parser: public*⟩+≡
```
   public :: parse_node_create_branch
```

⟨*Parser: procedures*⟩+≡
```
   subroutine parse_node_create_branch (node, rule)
     type(parse_node_t), pointer :: node
     type(syntax_rule_t), intent(in), target :: rule
     allocate (node)
     node%rule => rule
   end subroutine parse_node_create_branch
```

Append a sub-node. The sub-node must exist and be a valid target, otherwise nothing is done.

⟨*Parser: public*⟩+≡
```
   public :: parse_node_append_sub
```

⟨*Parser: procedures*⟩+≡
```
   subroutine parse_node_append_sub (node, sub)
     type(parse_node_t), intent(inout) :: node
     type(parse_node_t), pointer :: sub
```

```
         if (associated (sub)) then
            if (associated (node%sub_last)) then
               node%sub_last%next => sub
            else
               node%sub_first => sub
            end if
            node%sub_last => sub
         end if
      end subroutine parse_node_append_sub
```

For easy access, once the list is complete we count the number of sub-nodes. If there are no subnodes, the whole node is deleted.

⟨*Parser: procedures*⟩+≡
```
      subroutine parse_node_freeze_branch (node)
        type(parse_node_t), pointer :: node
        type(parse_node_t), pointer :: current
        node%n_sub = 0
        current => node%sub_first
        do while (associated (current))
           node%n_sub = node%n_sub + 1
           current => current%next
        end do
        if (node%n_sub == 0)  deallocate (node)
      end subroutine parse_node_freeze_branch
```

Replace the last subnode by a new target. Use with care, this invites to memory mismanagement.

⟨*Parser: public*⟩+≡
```
      public :: parse_node_replace_last_sub
```

⟨*Parser: procedures*⟩+≡
```
      subroutine parse_node_replace_last_sub (node, pn_target)
        type(parse_node_t), intent(inout), target :: node
        type(parse_node_t), intent(in), target :: pn_target
        type(parse_node_t), pointer :: current
        integer :: i
        select case (node%n_sub)
        case (1)
           node%sub_first => pn_target
        case (2:)
           current => node%sub_first
           do i = 1, node%n_sub - 2
              current => current%next
           end do
           current%next => pn_target
        case default
           call parse_node_write (node)
           call msg_bug ("'replace_last_sub' called for non-branch parse node")
        end select
        node%sub_last => pn_target
      end subroutine parse_node_replace_last_sub
```

3.4.5 Accessing nodes

Return the node contents. Check if pointers are associated. No check when accessing a sub-node; assume that `parse_node_n_sub` is always used for the upper bound.

The token extractor returns a pointer.

⟨*Parser: public*⟩+≡
```
public :: parse_node_get_rule_ptr
public :: parse_node_get_n_sub
public :: parse_node_get_sub_ptr
public :: parse_node_get_next_ptr
public :: parse_node_get_last_sub_ptr
```

⟨*Parser: procedures*⟩+≡
```
function parse_node_get_rule_ptr (node) result (rule)
  type(syntax_rule_t), pointer :: rule
  type(parse_node_t), intent(in), target :: node
  if (associated (node%rule)) then
     rule => node%rule
  else
     rule => null ()
     call parse_node_undefined (node, "rule")
  end if
end function parse_node_get_rule_ptr

function parse_node_get_n_sub (node) result (n)
  integer :: n
  type(parse_node_t), intent(in) :: node
  n = node%n_sub
end function parse_node_get_n_sub

function parse_node_get_sub_ptr (node, n, tag, required) result (sub)
  type(parse_node_t), pointer :: sub
  type(parse_node_t), intent(in), target :: node
  integer, intent(in), optional :: n
  character(*), intent(in), optional :: tag
  logical, intent(in), optional :: required
  integer :: i
  sub => node%sub_first
  if (present (n)) then
     do i = 2, n
        if (associated (sub)) then
           sub => sub%next
        else
           return
        end if
     end do
  end if
  call parse_node_check (sub, tag, required)
end function parse_node_get_sub_ptr

function parse_node_get_next_ptr (sub, n, tag, required) result (next)
  type(parse_node_t), pointer :: next
  type(parse_node_t), intent(in), target :: sub
  integer, intent(in), optional :: n
```

159

```
      character(*), intent(in), optional :: tag
      logical, intent(in), optional :: required
      integer :: i
      next => sub%next
      if (present (n)) then
         do i = 2, n
            if (associated (next)) then
               next => next%next
            else
               exit
            end if
         end do
      end if
      call parse_node_check (next, tag, required)
   end function parse_node_get_next_ptr

   function parse_node_get_last_sub_ptr (node, tag, required) result (sub)
      type(parse_node_t), pointer :: sub
      type(parse_node_t), intent(in), target :: node
      character(*), intent(in), optional :: tag
      logical, intent(in), optional :: required
      sub => node%sub_last
      call parse_node_check (sub, tag, required)
   end function parse_node_get_last_sub_ptr
```

⟨*Parser: procedures*⟩+≡
```
   subroutine parse_node_undefined (node, obj)
      type(parse_node_t), intent(in) :: node
      character(*), intent(in) :: obj
      call parse_node_write (node, 6)
      call msg_bug (" Parse-tree node: " // obj // " requested, but undefined")
   end subroutine parse_node_undefined
```

Check if a parse node has a particular tag, and if it is associated:

⟨*Parser: public*⟩+≡
```
   public :: parse_node_check
```
⟨*Parser: procedures*⟩+≡
```
   subroutine parse_node_check (node, tag, required)
      type(parse_node_t), pointer :: node
      character(*), intent(in), optional :: tag
      logical, intent(in), optional :: required
      if (associated (node)) then
         if (present (tag)) then
            if (parse_node_get_rule_key (node) /= tag) &
                 call parse_node_mismatch (tag, node)
         end if
      else
         if (present (required)) then
            if (required) &
                 call msg_bug (" Missing node, expected <" // tag // ">")
         end if
      end if
   end subroutine parse_node_check
```

This is called by a parse-tree scanner if the expected and the actual nodes do not match

⟨*Parser: public*⟩+≡
```
    public :: parse_node_mismatch
```

⟨*Parser: procedures*⟩+≡
```
    subroutine parse_node_mismatch (string, parse_node)
      character(*), intent(in) :: string
      type(parse_node_t), intent(in) :: parse_node
      call parse_node_write (parse_node)
      call msg_bug (" Syntax mismatch, expected <" // string // ">.")
    end subroutine parse_node_mismatch
```

The following functions are wrappers for extracting the token contents.

⟨*Parser: public*⟩+≡
```
    public :: parse_node_get_logical
    public :: parse_node_get_integer
    public :: parse_node_get_real
    public :: parse_node_get_cmplx
    public :: parse_node_get_string
    public :: parse_node_get_key
    public :: parse_node_get_rule_key
```

⟨*Parser: procedures*⟩+≡
```
    function parse_node_get_logical (node) result (lval)
      logical :: lval
      type(parse_node_t), intent(in), target :: node
      lval = token_get_logical (parse_node_get_token_ptr (node))
    end function parse_node_get_logical

    function parse_node_get_integer (node) result (ival)
      integer :: ival
      type(parse_node_t), intent(in), target :: node
      ival = token_get_integer (parse_node_get_token_ptr (node))
    end function parse_node_get_integer

    function parse_node_get_real (node) result (rval)
      real(default) :: rval
      type(parse_node_t), intent(in), target :: node
      rval = token_get_real (parse_node_get_token_ptr (node))
    end function parse_node_get_real

    function parse_node_get_cmplx (node) result (cval)
      complex(default) :: cval
      type(parse_node_t), intent(in), target :: node
      cval = token_get_cmplx (parse_node_get_token_ptr (node))
    end function parse_node_get_cmplx

    function parse_node_get_string (node) result (sval)
      type(string_t) :: sval
      type(parse_node_t), intent(in), target :: node
      sval = token_get_string (parse_node_get_token_ptr (node))
    end function parse_node_get_string
```

```
function parse_node_get_key (node) result (kval)
  type(string_t) :: kval
  type(parse_node_t), intent(in), target :: node
  kval = token_get_key (parse_node_get_token_ptr (node))
end function parse_node_get_key

function parse_node_get_rule_key (node) result (kval)
  type(string_t) :: kval
  type(parse_node_t), intent(in), target :: node
  kval = syntax_rule_get_key (parse_node_get_rule_ptr (node))
end function parse_node_get_rule_key

function parse_node_get_token_ptr (node) result (token)
  type(token_t), pointer :: token
  type(parse_node_t), intent(in), target :: node
  if (token_is_valid (node%token)) then
     token => node%token
  else
     call parse_node_undefined (node, "token")
  end if
end function parse_node_get_token_ptr
```

Return a MD5 sum for a parse node. The method is to write the node to a
scratch file and to evaluate the MD5 sum of that file.

⟨*Parser: public*⟩+≡

```
public :: parse_node_get_md5sum
```

⟨*Parser: procedures*⟩+≡

```
function parse_node_get_md5sum (pn) result (md5sum_pn)
  character(32) :: md5sum_pn
  type(parse_node_t), intent(in) :: pn
  integer :: u
  u = free_unit ()
  open (unit = u, status = "scratch", action = "readwrite")
  call parse_node_write_rec (pn, unit=u)
  rewind (u)
  md5sum_pn = md5sum (u)
  close (u)
end function parse_node_get_md5sum
```

3.4.6 The parse tree

The parse tree is a tree of nodes, where leaf nodes hold a valid token, while
branch nodes point to a list of sub-nodes.

⟨*Parser: public*⟩+≡

```
public :: parse_tree_t
```

⟨*Parser: types*⟩+≡

```
type :: parse_tree_t
   private
   type(parse_node_t), pointer :: root_node => null ()
```

```
    end type parse_tree_t
```

The parser. Its arguments are the parse tree (which should be empty initially),
the lexer (which should be already set up), the syntax table (which should be
valid), and the input stream. The input stream is completely parsed, using the
lexer setup and the syntax rules as given, and the parse tree is built accordingly.

If `check_eof` is absent or true, the parser will complain about trailing
garbage. Otherwise, it will ignore it.

By default, the input stream is matched against the top rule in the specified
syntax. If `key` is given, it is matched against the rule with the specified key
instead.

Failure at the top level means that no rule could match at all; in this case
the error message will show the top rule.

⟨*Parser: public*⟩+≡
```
    public :: parse_tree_init
```

⟨*Parser: procedures*⟩+≡
```
    subroutine parse_tree_init &
        (parse_tree, syntax, lexer, key, check_eof)
      type(parse_tree_t), intent(inout) :: parse_tree
      type(lexer_t), intent(inout) :: lexer
      type(syntax_t), intent(in), target :: syntax
      type(string_t), intent(in), optional :: key
      logical, intent(in), optional :: check_eof
      type(syntax_rule_t), pointer :: rule
      type(lexeme_t) :: lexeme
      type(parse_node_t), pointer :: node
      logical :: ok, check
      check = .true.;  if (present (check_eof)) check = check_eof
      call lexer_clear (lexer)
      if (present (key)) then
         rule => syntax_get_rule_ptr (syntax, key)
      else
         rule => syntax_get_top_rule_ptr (syntax)
      end if
      if (associated (rule)) then
         call parse_node_match_rule (node, rule, ok)
         if (ok) then
            parse_tree%root_node => node
         else
            call parse_error (rule, lexeme)
         end if
         if (check) then
            call lex (lexeme, lexer)
            if (.not. lexeme_is_eof (lexeme)) then
               call lexer_show_location (lexer)
               call msg_fatal (" Syntax error " &
                    // "(at or before the location indicated above)")
            end if
         end if
      else
         call msg_bug (" Parser failed because syntax is empty")
      end if
```

163

```
    contains
⟨Parser: internal subroutines of parse_tree_init⟩
    end subroutine parse_tree_init
```

The parser works recursively, following the rule tree, building the tree of nodes
on the fly. If the given rule matches, the node is filled on return. If not, the
node remains empty.

⟨*Parser: internal subroutines of* `parse_tree_init`⟩≡

```
    recursive subroutine parse_node_match_rule (node, rule, ok)
      type(parse_node_t), pointer :: node
      type(syntax_rule_t), intent(in), target :: rule
      logical, intent(out) :: ok
      logical, parameter :: debug = .false.
      integer :: type
      if (debug)  write (6, "(A)", advance="no") "Parsing rule: "
      if (debug)  call syntax_rule_write (rule, 6)
      node => null ()
      type = syntax_rule_get_type (rule)
      if (syntax_rule_is_atomic (rule)) then
         call lex (lexeme, lexer)
         if (debug)  write (6, "(A)", advance="no") "Token: "
         if (debug)  call lexeme_write (lexeme, 6)
         call parse_node_create_leaf (node, rule, lexeme)
         ok = associated (node)
         if (.not. ok)  call lexer_put_back (lexer, lexeme)
      else
         select case (type)
         case (S_ALTERNATIVE);  call parse_alternative (node, rule, ok)
         case (S_GROUP);        call parse_group (node, rule, ok)
         case (S_SEQUENCE);     call parse_sequence (node, rule, .false., ok)
         case (S_LIST);         call parse_sequence (node, rule, .true., ok)
         case (S_ARGS);         call parse_args (node, rule, ok)
         case (S_IGNORE);       call parse_ignore (node, ok)
         end select
      end if
      if (debug) then
         if (ok) then
            write (6, "(A)", advance="no") "Matched rule: "
         else
            write (6, "(A)", advance="no") "Failed rule: "
         end if
         call syntax_rule_write (rule)
         if (associated (node)) call parse_node_write (node)
      end if
    end subroutine parse_node_match_rule
```

Parse an alternative: try each case. If the match succeeds, the node has been
filled, so return. If nothing works, return failure.

⟨*Parser: internal subroutines of* `parse_tree_init`⟩+≡

```
    recursive subroutine parse_alternative (node, rule, ok)
      type(parse_node_t), pointer :: node
      type(syntax_rule_t), intent(in), target :: rule
      logical, intent(out) :: ok
      integer :: i
```

```
        do i = 1, syntax_rule_get_n_sub (rule)
           call parse_node_match_rule (node, syntax_rule_get_sub_ptr (rule, i), ok)
           if (ok)  return
        end do
        ok = .false.
     end subroutine parse_alternative
```

Parse a group: the first and third lexemes have to be the delimiters, the second one is parsed as the actual node, using now the child rule. If the first match fails, return with failure. If the other matches fail, issue an error, since we cannot lex back more than one item.

⟨*Parser: internal subroutines of* `parse_tree_init`⟩+≡

```
     recursive subroutine parse_group (node, rule, ok)
       type(parse_node_t), pointer :: node
       type(syntax_rule_t), intent(in), target :: rule
       logical, intent(out) :: ok
       type(string_t), dimension(2) :: delimiter
       delimiter = syntax_rule_get_delimiter (rule)
       call lex (lexeme, lexer)
       if (lexeme_get_string (lexeme) == delimiter(1)) then
          call parse_node_match_rule (node, syntax_rule_get_sub_ptr (rule, 1), ok)
          if (ok) then
             call lex (lexeme, lexer)
             if (lexeme_get_string (lexeme) == delimiter(2)) then
                ok = .true.
             else
                call parse_error (rule, lexeme)
             end if
          else
             call parse_error (rule, lexeme)
          end if
       else
          call lexer_put_back (lexer, lexeme)
          ok = .false.
       end if
     end subroutine parse_group
```

Parsing a sequence. The last rule element may be special: optional and/or repetitive. Each sub-node that matches is appended to the sub-node list of the parent node.

  If `sep` is true, we look for a separator after each element.

⟨*Parser: internal subroutines of* `parse_tree_init`⟩+≡

```
     recursive subroutine parse_sequence (node, rule, sep, ok)
       type(parse_node_t), pointer :: node
       type(syntax_rule_t), intent(in), target :: rule
       logical, intent(in) :: sep
       logical, intent(out) :: ok
       type(parse_node_t), pointer :: current
       integer :: i, n
       logical :: opt, rep, cont
       type(string_t) :: separator
       call parse_node_create_branch (node, rule)
       if (sep)  separator = syntax_rule_get_separator (rule)
       n = syntax_rule_get_n_sub (rule)
```

```
      opt = syntax_rule_last_optional (rule)
      rep = syntax_rule_last_repetitive (rule)
      ok = .true.
      cont = .true.
      SCAN_RULE: do i = 1, n
         call parse_node_match_rule &
              (current, syntax_rule_get_sub_ptr (rule, i), cont)
         if (cont) then
            call parse_node_append_sub (node, current)
            if (sep .and. (i<n .or. rep)) then
               call lex (lexeme, lexer)
               if (lexeme_get_string (lexeme) /= separator) then
                  call lexer_put_back (lexer, lexeme)
                  cont = .false.
                  exit SCAN_RULE
               end if
            end if
         else
            if (i == n .and. opt) then
               exit SCAN_RULE
            else if (i == 1) then
               ok = .false.
               exit SCAN_RULE
            else
               call parse_error (rule, lexeme)
            end if
         end if
      end do SCAN_RULE
      if (rep) then
         do while (cont)
            call parse_node_match_rule &
                 (current, syntax_rule_get_sub_ptr (rule, n), cont)
            if (cont) then
               call parse_node_append_sub (node, current)
               if (sep) then
                  call lex (lexeme, lexer)
                  if (lexeme_get_string (lexeme) /= separator) then
                     call lexer_put_back (lexer, lexeme)
                     cont = .false.
                  end if
               end if
            else
               if (sep)  call parse_error (rule, lexeme)
            end if
         end do
      end if
      call parse_node_freeze_branch (node)
    end subroutine parse_sequence
```

Argument list: We use the **parse_group** code, but call **parse_sequence** inside.

⟨*Parser: internal subroutines of* **parse_tree_init**⟩+≡

```
    recursive subroutine parse_args (node, rule, ok)
      type(parse_node_t), pointer :: node
      type(syntax_rule_t), intent(in), target :: rule
      logical, intent(out) :: ok
```

```
    type(string_t), dimension(2) :: delimiter
    delimiter = syntax_rule_get_delimiter (rule)
    call lex (lexeme, lexer)
    if (lexeme_get_string (lexeme) == delimiter(1)) then
        call parse_sequence (node, rule, .true., ok)
        if (ok) then
            call lex (lexeme, lexer)
            if (lexeme_get_string (lexeme) == delimiter(2)) then
                ok = .true.
            else
                call parse_error (rule, lexeme)
            end if
        else
            call parse_error (rule, lexeme)
        end if
    else
        call lexer_put_back (lexer, lexeme)
        ok = .false.
    end if
end subroutine parse_args
```

The IGNORE syntax reads one lexeme and discards it if it is numeric, logical or string/identifier (but not a keyword). This is a successful match. Otherwise, the match fails. The node pointer is returned disassociated in any case.

⟨*Parser: internal subroutines of* `parse_tree_init`⟩+≡

```
subroutine parse_ignore (node, ok)
  type(parse_node_t), pointer :: node
  logical, intent(out) :: ok
  call lex (lexeme, lexer)
  select case (lexeme_get_type (lexeme))
  case (T_NUMERIC, T_IDENTIFIER, T_QUOTED)
     ok = .true.
  case default
     ok = .false.
  end select
  node => null ()
end subroutine parse_ignore
```

If the match fails and we cannot step back:

⟨*Parser: internal subroutines of* `parse_tree_init`⟩+≡

```
subroutine parse_error (rule, lexeme)
  type(syntax_rule_t), intent(in) :: rule
  type(lexeme_t), intent(in) :: lexeme
  character(80) :: buffer
  integer :: u, iostat
  call lexer_show_location (lexer)
  u = free_unit ()
  open (u, status = "scratch")
  write (u, "(A)", advance="no")  "Expected syntax:"
  call syntax_rule_write (rule, u)
  write (u, "(A)", advance="no")  "Found token:"
  call lexeme_write (lexeme, u)
  rewind (u)
  do
     read (u, "(A)", iostat=iostat)  buffer
```

```
        if (iostat /= 0)  exit
        call msg_message (trim (buffer))
      end do
      call msg_fatal (" Syntax error " &
           // "(at or before the location indicated above)")
    end subroutine parse_error
```

The finalizer recursively deallocates all nodes and their contents. For each node, `parse_node_final` is called on the sub-nodes, which in turn deallocates the token or sub-node array contained within each of them. At the end, only the top node remains to be deallocated.

⟨*Parser: public*⟩+≡
```
    public :: parse_tree_final
```

⟨*Parser: procedures*⟩+≡
```
    subroutine parse_tree_final (parse_tree)
      type(parse_tree_t), intent(inout) :: parse_tree
      if (associated (parse_tree%root_node)) then
         call parse_node_final (parse_tree%root_node)
         deallocate (parse_tree%root_node)
      end if
    end subroutine parse_tree_final
```

Print the parse tree. Print one token per line, indented according to the depth of the node.

The `verbose` version includes type identifiers for the nodes.

⟨*Parser: public*⟩+≡
```
    public :: parse_tree_write
```

⟨*Parser: procedures*⟩+≡
```
    subroutine parse_tree_write (parse_tree, unit, verbose)
      type(parse_tree_t), intent(in) :: parse_tree
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer :: u
      logical :: short
      u = output_unit (unit);  if (u < 0)  return
      short = .true.;  if (present (verbose))  short = .not. verbose
      write (u, "(A)") "Parse tree:"
      if (associated (parse_tree%root_node)) then
         call parse_node_write_rec (parse_tree%root_node, unit, short, 1)
      else
         write (u, *) "[empty]"
      end if
    end subroutine parse_tree_write
```

This is a generic error that can be issued if the parse tree does not meet the expectaions of the parser. This most certainly indicates a bug.

⟨*Parser: public*⟩+≡
```
    public :: parse_tree_bug
```

⟨*Parser: procedures*⟩+≡
```
    subroutine parse_tree_bug (node, keys)
      type(parse_node_t), intent(in) :: node
```

```
      character(*), intent(in) :: keys
      call parse_node_write (node)
      call msg_bug (" Inconsistency in parse tree: expected " // keys)
    end subroutine parse_tree_bug
```

3.4.7 Access the parser

For scanning the parse tree we give access to the top node, as a node pointer.
Of course, there should be no write access.

⟨*Parser: public*⟩+≡
```
    public :: parse_tree_get_root_ptr
```

⟨*Parser: procedures*⟩+≡
```
    function parse_tree_get_root_ptr (parse_tree) result (node)
      type(parse_node_t), pointer :: node
      type(parse_tree_t), intent(in), target :: parse_tree
      node => parse_tree%root_node
    end function parse_tree_get_root_ptr
```

3.4.8 Applications

For a file of the form

```
process foo, bar
  <something>
process xyz
  <something>
```

get the `<something>` entry node for the first matching process tag. If no match-
ing entry is found, the node pointer remains unassociated.

⟨*Parser: public*⟩+≡
```
    public :: parse_tree_get_process_ptr
```

⟨*Parser: procedures*⟩+≡
```
    function parse_tree_get_process_ptr (parse_tree, process) result (node)
      type(parse_node_t), pointer :: node
      type(parse_tree_t), intent(in), target :: parse_tree
      type(string_t), intent(in) :: process
      type(parse_node_t), pointer :: node_root, node_process_def
      type(parse_node_t), pointer :: node_process_phs, node_process_list
      integer :: j
      node_root => parse_tree_get_root_ptr (parse_tree)
      if (associated (node_root)) then
         node_process_phs => parse_node_get_sub_ptr (node_root)
         SCAN_FILE: do while (associated (node_process_phs))
            node_process_def => parse_node_get_sub_ptr (node_process_phs)
            node_process_list => parse_node_get_sub_ptr (node_process_def, 2)
            do j = 1, parse_node_get_n_sub (node_process_list)
               if (parse_node_get_string &
                    (parse_node_get_sub_ptr (node_process_list, j)) &
                    == process) then
```

```
                node => parse_node_get_next_ptr (node_process_def)
                return
            end if
        end do
        node_process_phs => parse_node_get_next_ptr (node_process_phs)
    end do SCAN_FILE
else
    node => null ()
end if
end function parse_tree_get_process_ptr
```

3.4.9   Test the parser

⟨*Parser: public*⟩+≡
```
public :: parse_test
```

⟨*Parser: procedures*⟩+≡
```
subroutine parse_test
  use ifiles
  use lexers

  type(ifile_t) :: ifile
  type(syntax_t), target :: syntax
  type(lexer_t) :: lexer
  type(stream_t), target :: stream
  type(parse_tree_t), target :: parse_tree

  print "(A)", "Parser test"
  print *

  call ifile_append (ifile, "SEQ expr = term addition*")
  call ifile_append (ifile, "SEQ addition = plus_or_minus term")
  call ifile_append (ifile, "SEQ term = factor multiplication*")
  call ifile_append (ifile, "SEQ multiplication = times_or_over factor")
  call ifile_append (ifile, "SEQ factor = atom exponentiation*")
  call ifile_append (ifile, "SEQ exponentiation = '^' atom")
  call ifile_append (ifile, "ALT atom = real | delimited_expr")
  call ifile_append (ifile, "GRO delimited_expr = ( expr )")
  call ifile_append (ifile, "ALT plus_or_minus = '+' | '-'")
  call ifile_append (ifile, "ALT times_or_over = '*' | '/'")
  call ifile_append (ifile, "KEY '+'")
  call ifile_append (ifile, "KEY '-'")
  call ifile_append (ifile, "KEY '*'")
  call ifile_append (ifile, "KEY '/'")
  call ifile_append (ifile, "KEY '^'")
  call ifile_append (ifile, "REA real")

  print "(A)", "File contents (syntax definition):"
  call ifile_write (ifile)
  print "(A)", "EOF"
  print *

  call syntax_init (syntax, ifile)
```

```
   call ifile_final (ifile)
   call syntax_write (syntax)
   print *

   call lexer_init (lexer, &
        comment_chars = "", &
        quote_chars = "'", &
        quote_match = "'", &
        single_chars = "+-*/^()", &
        special_class = (/ "" /) , &
        keyword_list = syntax_get_keyword_list_ptr (syntax))
   call lexer_write_setup (lexer)
   print *

   call ifile_append (ifile, "(27+8^3-2/3)*(4+7)^2*99")
   print "(A)", "File contents (input file):"
   call ifile_write (ifile)
   print "(A)", "EOF"
   print *

   call stream_init (stream, ifile)
   call lexer_assign_stream (lexer, stream)
   call parse_tree_init (parse_tree, syntax, lexer)
   call stream_final (stream)
   call parse_tree_write (parse_tree)
   print *

   print "(A)", "Cleanup, everything should now be empty:"
   print *

   call parse_tree_final (parse_tree)
   call parse_tree_write (parse_tree)
   print *

   call lexer_final (lexer)
   call lexer_write_setup (lexer)
   print *

   call ifile_final (ifile)
   print "(A)", "File contents:"
   call ifile_write (ifile)
   print "(A)", "EOF"
   print *

   call syntax_final (syntax)
   call syntax_write (syntax)

end subroutine parse_test
```

Chapter 4

Physics library

This part consists of two modules:

constants Physical and mathematical parameters that never change. The file has been moved to the `src/misc` subdirectory of the `WHIZARD` project.

lorentz Define three-vectors, four-vectors and Lorentz transformations and common operations for them.

sm_physics Here, running functions are stored for special kinematical setup like running coupling constants, Catani-Seymour dipoles, or Sudakov factors.

4.1 Lorentz algebra

Define Lorentz vectors, three-vectors, boosts, and some functions to manipulate them.

To make maximum use of this, all functions, if possible, are declared elemental (or pure, if this is not possible).

⟨`lorentz.f90`⟩≡
  ⟨*File header*⟩

```
  module lorentz
```

  ⟨*Use kinds*⟩
```
    use constants, only: pi, twopi, degree !NODEP!
```
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
```

  ⟨*Standard module head*⟩

  ⟨*Lorentz: public*⟩

  ⟨*Lorentz: public operators*⟩

  ⟨*Lorentz: public functions*⟩

  ⟨*Lorentz: types*⟩

  ⟨*Lorentz: parameters*⟩

  ⟨*Lorentz: interfaces*⟩

```
  contains
```

  ⟨*Lorentz: subroutines*⟩
```
  end module lorentz
```

4.1.1 Three-vectors

First of all, let us introduce three-vectors in a trivial way. The functions and overloaded elementary operations clearly are too much overhead, but we like to keep the interface for three-vectors and four-vectors exactly parallel. By the way, we might attach a label to a vector by extending the type definition later.

⟨*Lorentz: public*⟩≡
```
    public :: vector3_t
```

⟨*Lorentz: types*⟩≡
```
    type :: vector3_t
       private
       real(default), dimension(3) :: p
    end type vector3_t
```

Output a vector

⟨*Lorentz: public*⟩+≡
```
    public :: vector3_write
```

⟨*Lorentz: subroutines*⟩≡
```
    subroutine vector3_write (p, unit)
      type(vector3_t), intent(in) :: p
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write(u, *) 'P = ', p%p
    end subroutine vector3_write
```

This is a three-vector with zero components
⟨*Lorentz: public*⟩+≡
```
    public :: vector3_null
```

⟨*Lorentz: parameters*⟩≡
```
    type(vector3_t), parameter :: vector3_null = &
        vector3_t ((/ 0._default, 0._default, 0._default /))
```

Canonical three-vector:
⟨*Lorentz: public*⟩+≡
```
    public :: vector3_canonical
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function vector3_canonical (k) result (p)
      type(vector3_t) :: p
      integer, intent(in) :: k
      p = vector3_null
      p%p(k) = 1
    end function vector3_canonical
```

A moving particle (k-axis, or arbitrary axis). Note that the function for the generic momentum cannot be elemental.
⟨*Lorentz: public*⟩+≡
```
    public :: vector3_moving
```

⟨*Lorentz: interfaces*⟩≡
```
    interface vector3_moving
       module procedure vector3_moving_canonical
       module procedure vector3_moving_generic
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function vector3_moving_canonical (p, k) result(q)
      type(vector3_t) :: q
      real(default), intent(in) :: p
      integer, intent(in) :: k
      q = vector3_null
      q%p(k) = p
    end function vector3_moving_canonical
    pure function vector3_moving_generic (p) result(q)
      real(default), dimension(3), intent(in) :: p
      type(vector3_t) :: q
      q%p = p
    end function vector3_moving_generic
```

Equality and inequality

⟨Lorentz: public operators⟩≡
```
    public :: operator(==), operator(/=)
```

⟨Lorentz: interfaces⟩+≡
```
    interface operator(==)
       module procedure vector3_eq
    end interface
    interface operator(/=)
       module procedure vector3_neq
    end interface
```

⟨Lorentz: subroutines⟩+≡
```
    elemental function vector3_eq (p, q) result (r)
      logical :: r
      type(vector3_t), intent(in) :: p,q
      r = all (p%p == q%p)
    end function vector3_eq
    elemental function vector3_neq (p, q) result (r)
      logical :: r
      type(vector3_t), intent(in) :: p,q
      r = any (p%p /= q%p)
    end function vector3_neq
```

Define addition and subtraction

⟨Lorentz: public operators⟩+≡
```
    public :: operator(+), operator(-)
```

⟨Lorentz: interfaces⟩+≡
```
    interface operator(+)
       module procedure add_vector3
    end interface
    interface operator(-)
       module procedure sub_vector3
    end interface
```

⟨Lorentz: subroutines⟩+≡
```
    elemental function add_vector3 (p, q) result (r)
      type(vector3_t) :: r
      type(vector3_t), intent(in) :: p,q
      r%p = p%p + q%p
    end function add_vector3
    elemental function sub_vector3 (p, q) result (r)
      type(vector3_t) :: r
      type(vector3_t), intent(in) :: p,q
      r%p = p%p - q%p
    end function sub_vector3
```

The multiplication sign is overloaded with scalar multiplication; similarly division:

⟨Lorentz: public operators⟩+≡
```
    public :: operator(*), operator(/)
```

⟨*Lorentz: interfaces*⟩+≡

```
interface operator(*)
   module procedure prod_integer_vector3, prod_vector3_integer
   module procedure prod_real_vector3, prod_vector3_real
end interface
interface operator(/)
   module procedure div_vector3_real, div_vector3_integer
end interface
```

⟨*Lorentz: subroutines*⟩+≡

```
elemental function prod_real_vector3 (s, p) result (q)
  type(vector3_t) :: q
  real(default), intent(in) :: s
  type(vector3_t), intent(in) :: p
  q%p = s * p%p
end function prod_real_vector3
elemental function prod_vector3_real (p, s) result (q)
  type(vector3_t) :: q
  real(default), intent(in) :: s
  type(vector3_t), intent(in) :: p
  q%p = s * p%p
end function prod_vector3_real
elemental function div_vector3_real (p, s) result (q)
  type(vector3_t) :: q
  real(default), intent(in) :: s
  type(vector3_t), intent(in) :: p
  q%p = p%p/s
end function div_vector3_real
elemental function prod_integer_vector3 (s, p) result (q)
  type(vector3_t) :: q
  integer, intent(in) :: s
  type(vector3_t), intent(in) :: p
  q%p = s * p%p
end function prod_integer_vector3
elemental function prod_vector3_integer (p, s) result (q)
  type(vector3_t) :: q
  integer, intent(in) :: s
  type(vector3_t), intent(in) :: p
  q%p = s * p%p
end function prod_vector3_integer
elemental function div_vector3_integer (p, s) result (q)
  type(vector3_t) :: q
  integer, intent(in) :: s
  type(vector3_t), intent(in) :: p
  q%p = p%p/s
end function div_vector3_integer
```

The multiplication sign can also indicate scalar products:

⟨*Lorentz: interfaces*⟩+≡

```
interface operator(*)
   module procedure prod_vector3
end interface
```

⟨*Lorentz: subroutines*⟩+≡

```
elemental function prod_vector3 (p, q) result (s)
```

```
      real(default) :: s
      type(vector3_t), intent(in) :: p,q
      s = dot_product (p%p, q%p)
    end function prod_vector3
```

⟨*Lorentz: public functions*⟩≡
```
    public :: cross_product
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface cross_product
       module procedure vector3_cross_product
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function vector3_cross_product (p, q) result (r)
      type(vector3_t) :: r
      type(vector3_t), intent(in) :: p,q
      integer :: i
      do i=1,3
         r%p(i) = dot_product (p%p, matmul(epsilon_three(i,:,:), q%p))
      end do
    end function vector3_cross_product
```

Exponentiation is defined only for integer powers. Odd powers mean take the square root; so p**1 is the length of p.

⟨*Lorentz: public operators*⟩+≡
```
    public :: operator(**)
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface operator(**)
       module procedure power_vector3
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function power_vector3 (p, e) result (s)
      real(default) :: s
      type(vector3_t), intent(in) :: p
      integer, intent(in) :: e
      s = dot_product (p%p, p%p)
      if (e/=2) then
         if (mod(e,2)==0) then
            s = s**(e/2)
         else
            s = sqrt(s)**e
         end if
      end if
    end function power_vector3
```

Finally, we need a negation.

⟨*Lorentz: interfaces*⟩+≡
```
    interface operator(-)
       module procedure negate_vector3
    end interface
```

177

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function negate_vector3 (p) result (q)
      type(vector3_t) :: q
      type(vector3_t), intent(in) :: p
      q%p = -p%p
    end function negate_vector3
```

The sum function can be useful:

⟨*Lorentz: public functions*⟩+≡
```
    public :: sum
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface sum
       module procedure sum_vector3
    end interface
```

There used to be a mask here, but the Intel compiler crashes with it

⟨*Lorentz: subroutines*⟩+≡
```
    pure function sum_vector3 (p) result (q)
      type(vector3_t) :: q
      type(vector3_t), dimension(:), intent(in) :: p
      integer :: i
      do i=1, 3
         q%p(i) = sum (p%p(i))
      end do
    end function sum_vector3
  !   pure function sum_vector3_mask (p, mask) result (q)
  !     type(vector3_t) :: q
  !     type(vector3_t), dimension(:), intent(in) :: p
  !     logical, dimension(:), intent(in) :: mask
  !     integer :: i
  !     do i=1, 3
  !        q%p(i) = sum (p%p(i), mask=mask)
  !     end do
  !   end function sum_vector3_mask
```

Any component:

⟨*Lorentz: public*⟩+≡
```
    public :: vector3_get_component
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function vector3_get_component (p, k) result (c)
      type(vector3_t), intent(in) :: p
      integer, intent(in) :: k
      real(default) :: c
      c = p%p(k)
    end function vector3_get_component
```

Extract all components. This is not elemental.

⟨*Lorentz: public*⟩+≡
```
    public :: vector3_get_components
```

⟨*Lorentz: subroutines*⟩+≡
```
pure function vector3_get_components (p) result (a)
  type(vector3_t), intent(in) :: p
  real(default), dimension(3) :: a
  a = p%p
end function vector3_get_components
```

This function returns the direction of a three-vector, i.e., a normalized three-vector

⟨*Lorentz: public functions*⟩+≡
```
public :: direction
```

⟨*Lorentz: interfaces*⟩+≡
```
interface direction
   module procedure vector3_get_direction
end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
elemental function vector3_get_direction (p) result (q)
  type(vector3_t) :: q
  type(vector3_t), intent(in) :: p
  q%p = p%p / p**1
end function vector3_get_direction
```

4.1.2 Four-vectors

In four-vectors the zero-component needs special treatment, therefore we do not use the standard operations. Sure, we pay for the extra layer of abstraction by losing efficiency; so we have to assume that the time-critical applications do not involve four-vector operations.

⟨*Lorentz: public*⟩+≡
```
public :: vector4_t
```

⟨*Lorentz: types*⟩+≡
```
type :: vector4_t
   private
   real(default), dimension(0:3) :: p
end type vector4_t
```

Output a vector

⟨*Lorentz: public*⟩+≡
```
public :: vector4_write
```

⟨*Lorentz: subroutines*⟩+≡
```
subroutine vector4_write (p, unit, show_mass)
  type(vector4_t), intent(in) :: p
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: show_mass
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  write(u, *) 'E = ', p%p(0)
  write(u, *) 'P = ', p%p(1:)
  if (present (show_mass)) then
     if (show_mass) &
```

```
            write (u, *) 'M = ', p**1
      end if
    end subroutine vector4_write
```

Binary I/O

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_write_raw
    public :: vector4_read_raw
```

⟨*Lorentz: subroutines*⟩+≡
```
    subroutine vector4_write_raw (p, u)
      type(vector4_t), intent(in) :: p
      integer, intent(in) :: u
      write (u) p%p
    end subroutine vector4_write_raw

    subroutine vector4_read_raw (p, u, iostat)
      type(vector4_t), intent(out) :: p
      integer, intent(in) :: u
      integer, intent(out), optional :: iostat
      read (u, iostat=iostat) p%p
    end subroutine vector4_read_raw
```

This is a four-vector with zero components

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_null
```

⟨*Lorentz: parameters*⟩+≡
```
    type(vector4_t), parameter :: vector4_null = &
         vector4_t ((/ 0._default, 0._default, 0._default, 0._default /))
```

Canonical four-vector:

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_canonical
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function vector4_canonical (k) result (p)
      type(vector4_t) :: p
      integer, intent(in) :: k
      p = vector4_null
      p%p(k) = 1
    end function vector4_canonical
```

A particle at rest:

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_at_rest
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function vector4_at_rest (m) result (p)
      type(vector4_t) :: p
      real(default), intent(in) :: m
      p = vector4_t ((/ m, 0._default, 0._default, 0._default /))
    end function vector4_at_rest
```

A moving particle (k-axis, or arbitrary axis)

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_moving
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface vector4_moving
       module procedure vector4_moving_canonical
       module procedure vector4_moving_generic
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function vector4_moving_canonical (E, p, k) result (q)
      type(vector4_t) :: q
      real(default), intent(in) :: E, p
      integer, intent(in) :: k
      q = vector4_at_rest(E)
      q%p(k) = p
    end function vector4_moving_canonical
    elemental function vector4_moving_generic (E, p) result (q)
      type(vector4_t) :: q
      real(default), intent(in) :: E
      type(vector3_t), intent(in) :: p
      q%p(0) = E
      q%p(1:) = p%p
    end function vector4_moving_generic
```

Equality and inequality

⟨*Lorentz: interfaces*⟩+≡
```
    interface operator(==)
       module procedure vector4_eq
    end interface
    interface operator(/=)
       module procedure vector4_neq
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function vector4_eq (p, q) result (r)
      logical :: r
      type(vector4_t), intent(in) :: p,q
      r = all (p%p == q%p)
    end function vector4_eq
    elemental function vector4_neq (p, q) result (r)
      logical :: r
      type(vector4_t), intent(in) :: p,q
      r = any (p%p /= q%p)
    end function vector4_neq
```

Addition and subtraction:

⟨*Lorentz: interfaces*⟩+≡
```
    interface operator(+)
       module procedure add_vector4
    end interface
    interface operator(-)
       module procedure sub_vector4
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function add_vector4 (p,q) result (r)
      type(vector4_t) :: r
      type(vector4_t), intent(in) :: p,q
      r%p = p%p + q%p
    end function add_vector4
    elemental function sub_vector4 (p,q) result (r)
      type(vector4_t) :: r
      type(vector4_t), intent(in) :: p,q
      r%p = p%p - q%p
    end function sub_vector4
```

We also need scalar multiplication and division:

⟨*Lorentz: interfaces*⟩+≡
```
    interface operator(*)
       module procedure prod_real_vector4, prod_vector4_real
       module procedure prod_integer_vector4, prod_vector4_integer
    end interface
    interface operator(/)
       module procedure div_vector4_real
       module procedure div_vector4_integer
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function prod_real_vector4 (s, p) result (q)
      type(vector4_t) :: q
      real(default), intent(in) :: s
      type(vector4_t), intent(in) :: p
      q%p = s * p%p
    end function prod_real_vector4
    elemental function prod_vector4_real (p, s) result (q)
      type(vector4_t) :: q
      real(default), intent(in) :: s
      type(vector4_t), intent(in) :: p
      q%p = s * p%p
    end function prod_vector4_real
    elemental function div_vector4_real (p, s) result (q)
      type(vector4_t) :: q
      real(default), intent(in) :: s
      type(vector4_t), intent(in) :: p
      q%p = p%p/s
    end function div_vector4_real
    elemental function prod_integer_vector4 (s, p) result (q)
      type(vector4_t) :: q
      integer, intent(in) :: s
      type(vector4_t), intent(in) :: p
      q%p = s * p%p
    end function prod_integer_vector4
    elemental function prod_vector4_integer (p, s) result (q)
      type(vector4_t) :: q
      integer, intent(in) :: s
      type(vector4_t), intent(in) :: p
      q%p = s * p%p
    end function prod_vector4_integer
```

```
      elemental function div_vector4_integer (p, s) result (q)
        type(vector4_t) :: q
        integer, intent(in) :: s
        type(vector4_t), intent(in) :: p
        q%p = p%p/s
      end function div_vector4_integer
```

Scalar products and squares in the Minkowski sense:

⟨*Lorentz: interfaces*⟩+≡
```
      interface operator(*)
         module procedure prod_vector4
      end interface
      interface operator(**)
         module procedure power_vector4
      end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
      elemental function prod_vector4 (p, q) result (s)
        real(default) :: s
        type(vector4_t), intent(in) :: p,q
        s = p%p(0)*q%p(0) - dot_product(p%p(1:), q%p(1:))
      end function prod_vector4
```

The power operation for four-vectors is signed, i.e., `p**1` is positive for timelike and negative for spacelike vectors. Note that `(p**1)**2` is not necessarily equal to `p**2`.

⟨*Lorentz: subroutines*⟩+≡
```
      elemental function power_vector4 (p, e) result (s)
        real(default) :: s
        type(vector4_t), intent(in) :: p
        integer, intent(in) :: e
        s = p*p
        if (e/=2) then
           if (mod(e,2)==0) then
              s = s**(e/2)
           elseif (s>=0) then
              s = sqrt(s)**e
           else
              s = -(sqrt(abs(s))**e)
           end if
        end if
      end function power_vector4
```

Finally, we introduce a negation

⟨*Lorentz: interfaces*⟩+≡
```
      interface operator(-)
         module procedure negate_vector4
      end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
      elemental function negate_vector4 (p) result (q)
        type(vector4_t) :: q
        type(vector4_t), intent(in) :: p
```

```
    q%p = -p%p
  end function negate_vector4
```

The sum function can be useful:

⟨*Lorentz: interfaces*⟩+≡
```
    interface sum
       module procedure sum_vector4
    end interface
```

There used to be a mask here, but the Intel compiler crashes with it

⟨*Lorentz: subroutines*⟩+≡
```
    pure function sum_vector4 (p) result (q)
      type(vector4_t) :: q
      type(vector4_t), dimension(:), intent(in) :: p
      integer :: i
      do i=0, 3
         q%p(i) = sum (p%p(i))
      end do
    end function sum_vector4
  !   pure function sum_vector4_mask (p, mask) result (q)
  !     type(vector4_t) :: q
  !     type(vector4_t), dimension(:), intent(in) :: p
  !     logical, dimension(:), intent(in) :: mask
  !     integer :: i
  !     do i=0, 3
  !        q%p(i) = sum (p%p(i), mask=mask)
  !     end do
  !   end function sum_vector4_mask
```

4.1.3   Conversions

Any component:

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_get_component
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function vector4_get_component (p, k) result (c)
      real(default) :: c
      type(vector4_t), intent(in) :: p
      integer, intent(in) :: k
      c = p%p(k)
    end function vector4_get_component
```

Extract all components. This is not elemental.

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_get_components
```

⟨*Lorentz: subroutines*⟩+≡
```
    pure function vector4_get_components (p) result (a)
      real(default), dimension(0:3) :: a
      type(vector4_t), intent(in) :: p
      a = p%p
```

```
    end function vector4_get_components
```

This function returns the space part of a four-vector, such that we can apply three-vector operations on it:

⟨*Lorentz: public functions*⟩+≡
```
    public :: space_part
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface space_part
       module procedure vector4_get_space_part
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function vector4_get_space_part (p) result (q)
      type(vector3_t) :: q
      type(vector4_t), intent(in) :: p
      q%p = p%p(1:)
    end function vector4_get_space_part
```

This function returns the direction of a four-vector, i.e., a normalized three-vector

⟨*Lorentz: interfaces*⟩+≡
```
    interface direction
       module procedure vector4_get_direction
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function vector4_get_direction (p) result (q)
      type(vector3_t) :: q
      type(vector4_t), intent(in) :: p
      q%p = p%p(1:)
      q = q / q**1
    end function vector4_get_direction
```

This function returns the four-vector as an ordinary array. A second version for an array of four-vectors.

⟨*Lorentz: public functions*⟩+≡
```
    public :: array_from_vector4
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface array_from_vector4
       module procedure array_from_vector4_1
       module procedure array_from_vector4_2
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    pure function array_from_vector4_1 (p) result (a)
      type(vector4_t), intent(in) :: p
      real(default), dimension(0:3) :: a
      a = p%p
    end function array_from_vector4_1

    pure function array_from_vector4_2 (p) result (a)
      type(vector4_t), dimension(:), intent(in) :: p
```

```
      real(default), dimension(0:3, size(p)) :: a
      integer :: i
      forall (i=1:size(p))
         a(0:3,i) = p(i)%p
      end forall
   end function array_from_vector4_2
```

4.1.4 Angles

Return the angles in a canonical system. The angle ϕ is defined between $0 \leq \phi < 2\pi$. In degenerate cases, return zero.

⟨*Lorentz: public functions*⟩+≡
```
   public :: azimuthal_angle
```

⟨*Lorentz: interfaces*⟩+≡
```
   interface azimuthal_angle
      module procedure vector3_azimuthal_angle
      module procedure vector4_azimuthal_angle
   end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
   elemental function vector3_azimuthal_angle (p) result (phi)
     real(default) :: phi
     type(vector3_t), intent(in) :: p
     if (any(p%p(1:2)/=0)) then
        phi = atan2(p%p(2), p%p(1))
        if (phi < 0) phi = phi + twopi
     else
        phi = 0
     end if
   end function vector3_azimuthal_angle
   elemental function vector4_azimuthal_angle (p) result (phi)
     real(default) :: phi
     type(vector4_t), intent(in) :: p
     phi = vector3_azimuthal_angle (space_part (p))
   end function vector4_azimuthal_angle
```

Azimuthal angle in degrees

⟨*Lorentz: public functions*⟩+≡
```
   public :: azimuthal_angle_deg
```

⟨*Lorentz: interfaces*⟩+≡
```
   interface azimuthal_angle_deg
      module procedure vector3_azimuthal_angle_deg
      module procedure vector4_azimuthal_angle_deg
   end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
   elemental function vector3_azimuthal_angle_deg (p) result (phi)
     real(default) :: phi
     type(vector3_t), intent(in) :: p
     phi = vector3_azimuthal_angle (p) / degree
   end function vector3_azimuthal_angle_deg
   elemental function vector4_azimuthal_angle_deg (p) result (phi)
```

```
      real(default) :: phi
      type(vector4_t), intent(in) :: p
      phi = vector4_azimuthal_angle (p) / degree
    end function vector4_azimuthal_angle_deg
```

The azimuthal distance of two vectors. This is the difference of the azimuthal angles, but cannot be larger than π: The result is between $-\pi < \Delta\phi \leq \pi$.

⟨*Lorentz: public functions*⟩+≡
```
    public :: azimuthal_distance
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface azimuthal_distance
       module procedure vector3_azimuthal_distance
       module procedure vector4_azimuthal_distance
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function vector3_azimuthal_distance (p, q) result (dphi)
      real(default) :: dphi
      type(vector3_t), intent(in) :: p,q
      dphi = vector3_azimuthal_angle (q) - vector3_azimuthal_angle (p)
      if (dphi <= -pi) then
         dphi = dphi + twopi
      else if (dphi > pi) then
         dphi = dphi - twopi
      end if
    end function vector3_azimuthal_distance
    elemental function vector4_azimuthal_distance (p, q) result (dphi)
      real(default) :: dphi
      type(vector4_t), intent(in) :: p,q
      dphi = vector3_azimuthal_distance &
           (space_part (p), space_part (q))
    end function vector4_azimuthal_distance
```

The same in degrees:

⟨*Lorentz: public functions*⟩+≡
```
    public :: azimuthal_distance_deg
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface azimuthal_distance_deg
       module procedure vector3_azimuthal_distance_deg
       module procedure vector4_azimuthal_distance_deg
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function vector3_azimuthal_distance_deg (p, q) result (dphi)
      real(default) :: dphi
      type(vector3_t), intent(in) :: p,q
      dphi = vector3_azimuthal_distance (p, q) / degree
    end function vector3_azimuthal_distance_deg
    elemental function vector4_azimuthal_distance_deg (p, q) result (dphi)
      real(default) :: dphi
      type(vector4_t), intent(in) :: p,q
      dphi = vector4_azimuthal_distance (p, q) / degree
    end function vector4_azimuthal_distance_deg
```

The polar angle is defined $0 \leq \theta \leq \pi$. Note that `ATAN2` has the reversed order of arguments: `ATAN2(Y,X)`. Here, x is the 3-component while y is the transverse momentum which is always nonnegative. Therefore, the result is nonnegative as well.

⟨*Lorentz: public functions*⟩+≡
```
    public :: polar_angle
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface polar_angle
       module procedure polar_angle_vector3
       module procedure polar_angle_vector4
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function polar_angle_vector3 (p) result (theta)
      real(default) :: theta
      type(vector3_t), intent(in) :: p
      if (any(p%p/=0)) then
         theta = atan2 (sqrt(p%p(1)**2 + p%p(2)**2), p%p(3))
      else
         theta = 0
      end if
    end function polar_angle_vector3
    elemental function polar_angle_vector4 (p) result (theta)
      real(default) :: theta
      type(vector4_t), intent(in) :: p
      theta = polar_angle (space_part (p))
    end function polar_angle_vector4
```

This is the cosine of the polar angle: $-1 \leq \cos\theta \leq 1$.

⟨*Lorentz: public functions*⟩+≡
```
    public :: polar_angle_ct
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface polar_angle_ct
       module procedure polar_angle_ct_vector3
       module procedure polar_angle_ct_vector4
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function polar_angle_ct_vector3 (p) result (ct)
      real(default) :: ct
      type(vector3_t), intent(in) :: p
      if (any(p%p/=0)) then
         ct = p%p(3) / p**1
      else
         ct = 1
      end if
    end function polar_angle_ct_vector3
    elemental function polar_angle_ct_vector4 (p) result (ct)
      real(default) :: ct
      type(vector4_t), intent(in) :: p
      ct = polar_angle_ct (space_part (p))
    end function polar_angle_ct_vector4
```

The polar angle in degrees.

⟨Lorentz: public functions⟩+≡
```
    public :: polar_angle_deg
```

⟨Lorentz: interfaces⟩+≡
```
    interface polar_angle_deg
       module procedure polar_angle_deg_vector3
       module procedure polar_angle_deg_vector4
    end interface
```

⟨Lorentz: subroutines⟩+≡
```
    elemental function polar_angle_deg_vector3 (p) result (theta)
      real(default) :: theta
      type(vector3_t), intent(in) :: p
      theta = polar_angle (p) / degree
    end function polar_angle_deg_vector3
    elemental function polar_angle_deg_vector4 (p) result (theta)
      real(default) :: theta
      type(vector4_t), intent(in) :: p
      theta = polar_angle (p) / degree
    end function polar_angle_deg_vector4
```

This is the angle enclosed between two three-momenta. If one of the momenta is zero, we return an angle of zero. The range of the result is $0 \leq \theta \leq \pi$. If there is only one argument, take the positive z axis as reference.

⟨Lorentz: public functions⟩+≡
```
    public :: enclosed_angle
```

⟨Lorentz: interfaces⟩+≡
```
    interface enclosed_angle
       module procedure enclosed_angle_vector3
       module procedure enclosed_angle_vector4
    end interface
```

⟨Lorentz: subroutines⟩+≡
```
    elemental function enclosed_angle_vector3 (p, q) result (theta)
      real(default) :: theta
      type(vector3_t), intent(in) :: p, q
      theta = acos (enclosed_angle_ct (p, q))
    end function enclosed_angle_vector3
    elemental function enclosed_angle_vector4 (p, q) result (theta)
      real(default) :: theta
      type(vector4_t), intent(in) :: p, q
      theta = enclosed_angle (space_part (p), space_part (q))
    end function enclosed_angle_vector4
```

The cosine of the enclosed angle.

⟨Lorentz: public functions⟩+≡
```
    public :: enclosed_angle_ct
```

⟨Lorentz: interfaces⟩+≡
```
    interface enclosed_angle_ct
       module procedure enclosed_angle_ct_vector3
       module procedure enclosed_angle_ct_vector4
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function enclosed_angle_ct_vector3 (p, q) result (ct)
      real(default) :: ct
      type(vector3_t), intent(in) :: p, q
      if (any(p%p/=0).and.any(q%p/=0)) then
         ct = p*q / (p**1 * q**1)
         if (ct>1) then
            ct = 1
         else if (ct<-1) then
            ct = -1
         end if
      else
         ct = 1
      end if
    end function enclosed_angle_ct_vector3
    elemental function enclosed_angle_ct_vector4 (p, q) result (ct)
      real(default) :: ct
      type(vector4_t), intent(in) :: p, q
      ct = enclosed_angle_ct (space_part (p), space_part (q))
    end function enclosed_angle_ct_vector4
```

The enclosed angle in degrees.

⟨*Lorentz: public functions*⟩+≡
```
    public :: enclosed_angle_deg
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface enclosed_angle_deg
       module procedure enclosed_angle_deg_vector3
       module procedure enclosed_angle_deg_vector4
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function enclosed_angle_deg_vector3 (p, q) result (theta)
      real(default) :: theta
      type(vector3_t), intent(in) :: p, q
      theta = enclosed_angle (p, q) / degree
    end function enclosed_angle_deg_vector3
    elemental function enclosed_angle_deg_vector4 (p, q) result (theta)
      real(default) :: theta
      type(vector4_t), intent(in) :: p, q
      theta = enclosed_angle (p, q) / degree
    end function enclosed_angle_deg_vector4
```

The polar angle of the first momentum w.r.t. the second momentum, evaluated in the rest frame of the second momentum. If the second four-momentum is not timelike, return zero.

⟨*Lorentz: public functions*⟩+≡
```
    public :: enclosed_angle_rest_frame
    public :: enclosed_angle_ct_rest_frame
    public :: enclosed_angle_deg_rest_frame
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface enclosed_angle_rest_frame
       module procedure enclosed_angle_rest_frame_vector4
```

```
      end interface
      interface enclosed_angle_ct_rest_frame
         module procedure enclosed_angle_ct_rest_frame_vector4
      end interface
      interface enclosed_angle_deg_rest_frame
         module procedure enclosed_angle_deg_rest_frame_vector4
      end interface
```
⟨*Lorentz: subroutines*⟩+≡
```
      elemental function enclosed_angle_rest_frame_vector4 (p, q) result (theta)
        type(vector4_t), intent(in) :: p, q
        real(default) :: theta
        theta = acos (enclosed_angle_ct_rest_frame (p, q))
      end function enclosed_angle_rest_frame_vector4
      elemental function enclosed_angle_ct_rest_frame_vector4 (p, q) result (ct)
        type(vector4_t), intent(in) :: p, q
        real(default) :: ct
        if (invariant_mass(q) > 0) then
           ct = enclosed_angle_ct ( &
                space_part (boost(-q, invariant_mass (q)) * p), &
                space_part (q))
        else
           ct = 1
        end if
      end function enclosed_angle_ct_rest_frame_vector4
      elemental function enclosed_angle_deg_rest_frame_vector4 (p, q) &
           result (theta)
        type(vector4_t), intent(in) :: p, q
        real(default) :: theta
        theta = enclosed_angle_rest_frame (p, q) / degree
      end function enclosed_angle_deg_rest_frame_vector4
```

4.1.5   More kinematical functions (some redundant)

The scalar transverse momentum (assuming the z axis is longitudinal)

⟨*Lorentz: public functions*⟩+≡
```
      public :: transverse_part
```
⟨*Lorentz: interfaces*⟩+≡
```
      interface transverse_part
         module procedure transverse_part_vector4
      end interface
```
⟨*Lorentz: subroutines*⟩+≡
```
      elemental function transverse_part_vector4 (p) result (pT)
        real(default) :: pT
        type(vector4_t), intent(in) :: p
        pT = sqrt(p%p(1)**2 + p%p(2)**2)
      end function transverse_part_vector4
```

The scalar longitudinal momentum (assuming the z axis is longitudinal). Identical to `momentum_z_component`.

⟨*Lorentz: public functions*⟩+≡
```
      public :: longitudinal_part
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface longitudinal_part
       module procedure longitudinal_part_vector4
    end interface
```
⟨*Lorentz: subroutines*⟩+≡
```
    elemental function longitudinal_part_vector4 (p) result (pL)
      real(default) :: pL
      type(vector4_t), intent(in) :: p
      pL = p%p(3)
    end function longitudinal_part_vector4
```

Absolute value of three-momentum

⟨*Lorentz: public functions*⟩+≡
```
    public :: space_part_norm
```
⟨*Lorentz: interfaces*⟩+≡
```
    interface space_part_norm
       module procedure space_part_norm_vector4
    end interface
```
⟨*Lorentz: subroutines*⟩+≡
```
    elemental function space_part_norm_vector4 (p) result (p3)
      real(default) :: p3
      type(vector4_t), intent(in) :: p
      p3 = sqrt (p%p(1)**2 + p%p(2)**2 + p%p(3)**2)
    end function space_part_norm_vector4
```

The energy (the zeroth component)

⟨*Lorentz: public functions*⟩+≡
```
    public :: energy
```
⟨*Lorentz: interfaces*⟩+≡
```
    interface energy
       module procedure energy_vector4
       module procedure energy_vector3
       module procedure energy_real
    end interface
```
⟨*Lorentz: subroutines*⟩+≡
```
    elemental function energy_vector4 (p) result (E)
      real(default) :: E
      type(vector4_t), intent(in) :: p
      E = p%p(0)
    end function energy_vector4
```

Alternative: The energy corresponding to a given momentum and mass. If the mass is omitted, it is zero

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function energy_vector3 (p, mass) result (E)
      real(default) :: E
      type(vector3_t), intent(in) :: p
      real(default), intent(in), optional :: mass
      if (present (mass)) then
         E = sqrt (p**2 + mass**2)
```

```
      else
         E = p**1
      end if
   end function energy_vector3

   elemental function energy_real (p, mass) result (E)
     real(default) :: E
     real(default), intent(in) :: p
     real(default), intent(in), optional :: mass
     if (present (mass)) then
        E = sqrt (p**2 + mass**2)
     else
        E = abs (p)
     end if
   end function energy_real
```

The invariant mass of four-momenta. Zero for lightlike, negative for spacelike momenta.

⟨*Lorentz: public functions*⟩+≡
```
   public :: invariant_mass
```

⟨*Lorentz: interfaces*⟩+≡
```
   interface invariant_mass
      module procedure invariant_mass_vector4
   end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
   elemental function invariant_mass_vector4 (p) result (m)
     real(default) :: m
     type(vector4_t), intent(in) :: p
     real(default) :: msq
     msq = p*p
     if (msq >= 0) then
        m = sqrt (msq)
     else
        m = - sqrt (abs (msq))
     end if
   end function invariant_mass_vector4
```

The invariant mass squared. Zero for lightlike, negative for spacelike momenta.

⟨*Lorentz: public functions*⟩+≡
```
   public :: invariant_mass_squared
```

⟨*Lorentz: interfaces*⟩+≡
```
   interface invariant_mass_squared
      module procedure invariant_mass_squared_vector4
   end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
   elemental function invariant_mass_squared_vector4 (p) result (msq)
     real(default) :: msq
     type(vector4_t), intent(in) :: p
     msq = p*p
   end function invariant_mass_squared_vector4
```

The transverse mass. If the mass squared is negative, this value also is negative.

⟨*Lorentz: public functions*⟩+≡
```
    public :: transverse_mass
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface transverse_mass
       module procedure transverse_mass_vector4
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function transverse_mass_vector4 (p) result (m)
      real(default) :: m
      type(vector4_t), intent(in) :: p
      real(default) :: msq
      msq = p%p(0)**2 - p%p(1)**2 - p%p(2)**2
      if (msq >= 0) then
         m = sqrt (msq)
      else
         m = - sqrt (abs (msq))
      end if
    end function transverse_mass_vector4
```

The rapidity (defined if particle is massive or $p_\perp > 0$)

⟨*Lorentz: public functions*⟩+≡
```
    public :: rapidity
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface rapidity
       module procedure rapidity_vector4
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function rapidity_vector4 (p) result (y)
      real(default) :: y
      type(vector4_t), intent(in) :: p
      y = .5 * log( (energy (p) + longitudinal_part (p)) &
           &        /(energy (p) - longitudinal_part (p)))
    end function rapidity_vector4
```

The pseudorapidity (defined if $p_\perp > 0$)

⟨*Lorentz: public functions*⟩+≡
```
    public :: pseudorapidity
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface pseudorapidity
       module procedure pseudorapidity_vector4
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function pseudorapidity_vector4 (p) result (eta)
      real(default) :: eta
      type(vector4_t), intent(in) :: p
      eta = -log( tan (.5 * polar_angle (p)))
    end function pseudorapidity_vector4
```

The rapidity distance (defined if both $p_\perp > 0$)

⟨Lorentz: public functions⟩+≡
```
    public :: rapidity_distance
```

⟨Lorentz: interfaces⟩+≡
```
    interface rapidity_distance
       module procedure rapidity_distance_vector4
    end interface
```

⟨Lorentz: subroutines⟩+≡
```
    elemental function rapidity_distance_vector4 (p, q) result (dy)
      type(vector4_t), intent(in) :: p, q
      real(default) :: dy
      dy = rapidity (q) - rapidity (p)
    end function rapidity_distance_vector4
```

The pseudorapidity distance (defined if both $p_\perp > 0$)

⟨Lorentz: public functions⟩+≡
```
    public :: pseudorapidity_distance
```

⟨Lorentz: interfaces⟩+≡
```
    interface pseudorapidity_distance
       module procedure pseudorapidity_distance_vector4
    end interface
```

⟨Lorentz: subroutines⟩+≡
```
    elemental function pseudorapidity_distance_vector4 (p, q) result (deta)
      real(default) :: deta
      type(vector4_t), intent(in) :: p, q
      deta = pseudorapidity (q) - pseudorapidity (p)
    end function pseudorapidity_distance_vector4
```

The distance on the $\eta - \phi$ cylinder:

⟨Lorentz: public functions⟩+≡
```
    public :: eta_phi_distance
```

⟨Lorentz: interfaces⟩+≡
```
    interface eta_phi_distance
       module procedure eta_phi_distance_vector4
    end interface
```

⟨Lorentz: subroutines⟩+≡
```
    elemental function eta_phi_distance_vector4 (p, q) result (dr)
      type(vector4_t), intent(in) :: p, q
      real(default) :: dr
      dr = sqrt ( &
           pseudorapidity_distance (p, q)**2 &
           + azimuthal_distance (p, q)**2)
    end function eta_phi_distance_vector4
```

4.1.6 Lorentz transformations

⟨Lorentz: public⟩+≡
```
    public :: lorentz_transformation_t
```

⟨*Lorentz: types*⟩+≡
```
type :: lorentz_transformation_t
   private
   real(default), dimension(0:3, 0:3) :: L
end type lorentz_transformation_t
```

Output:

⟨*Lorentz: public*⟩+≡
```
public :: lorentz_transformation_write
```

⟨*Lorentz: subroutines*⟩+≡
```
subroutine lorentz_transformation_write (L, unit)
  type(lorentz_transformation_t), intent(in) :: L
  integer, intent(in), optional :: unit
  integer :: u
  integer :: i
  u = output_unit (unit);  if (u < 0)  return
  write (u, *) 'Lorentz transformation:'
  write (u, *) 'L00:'
  write (u, *) L%L(0,0)
  write (u, *) 'L0j:', L%L(0,1:)
  write (u, *) 'Li0, Lij:'
  do i = 1, 3
     write (u, *) L%L(i,0)
     write (u, *) '     ', L%L(i,1:)
  end do
end subroutine lorentz_transformation_write
```

Extract all components:

⟨*Lorentz: public*⟩+≡
```
public :: lorentz_transformation_get_components
```

⟨*Lorentz: subroutines*⟩+≡
```
pure function lorentz_transformation_get_components (L) result (a)
  type(lorentz_transformation_t), intent(in) :: L
  real(default), dimension(0:3,0:3) :: a
  a = L%L
end function lorentz_transformation_get_components
```

4.1.7 Functions of Lorentz transformations

For the inverse, we make use of the fact that $\Lambda^{\mu\nu}\Lambda_{\mu\rho} = \delta^{\nu}_{\rho}$. So, lowering the indices and transposing is sufficient.

⟨*Lorentz: public functions*⟩+≡
```
public :: inverse
```

⟨*Lorentz: interfaces*⟩+≡
```
interface inverse
   module procedure lorentz_transformation_inverse
end interface
```

⟨*Lorentz: subroutines*⟩+≡

```fortran
    elemental function lorentz_transformation_inverse (L) result (IL)
      type(lorentz_transformation_t) :: IL
      type(lorentz_transformation_t), intent(in) :: L
      IL%L(0,0) = L%L(0,0)
      IL%L(0,1:) = -L%L(1:,0)
      IL%L(1:,0) = -L%L(0,1:)
      IL%L(1:,1:) = transpose(L%L(1:,1:))
    end function lorentz_transformation_inverse
```

4.1.8   Invariants

These are used below. The first array index is varying fastest in FORTRAN; therefore the extra minus in the odd-rank tensor epsilon.

⟨*Lorentz: parameters*⟩+≡

```fortran
    integer, dimension(3,3), parameter :: delta_three = &
        & reshape( source = (/ 1,0,0, 0,1,0, 0,0,1 /), &
        &          shape  = (/3,3/) )
    integer, dimension(3,3,3), parameter :: epsilon_three = &
        & reshape( source = (/ 0, 0,0,  0,0,-1,   0,1,0,&
        &                      0, 0,1,  0,0, 0,  -1,0,0,&
        &                      0,-1,0,  1,0, 0,   0,0,0 /),&
        &          shape = (/3,3,3/) )
```

This could be of some use:

⟨*Lorentz: public*⟩+≡

```fortran
    public :: identity
```

⟨*Lorentz: parameters*⟩+≡

```fortran
    type(lorentz_transformation_t), parameter :: &
        & identity = &
        & lorentz_transformation_t ( &
        & reshape( source = (/ 1._default, 0._default, 0._default, 0._default, &
        &                      0._default, 1._default, 0._default, 0._default, &
        &                      0._default, 0._default, 1._default, 0._default, &
        &                      0._default, 0._default, 0._default, 1._default /),&
        &          shape = (/ 4,4 /) ) )
```

⟨*Lorentz: public*⟩+≡

```fortran
    public :: space_reflection
```

⟨*Lorentz: parameters*⟩+≡

```fortran
    type(lorentz_transformation_t), parameter :: &
        & space_reflection = &
        & lorentz_transformation_t ( &
        & reshape( source = (/ 1._default, 0._default, 0._default, 0._default, &
        &                      0._default,-1._default, 0._default, 0._default, &
        &                      0._default, 0._default,-1._default, 0._default, &
        &                      0._default, 0._default, 0._default,-1._default /),&
        &          shape = (/ 4,4 /) ) )
```

4.1.9 Boosts

We build Lorentz transformations from boosts and rotations. In both cases we can supply a three-vector which defines the axis and (hyperbolic) angle. For a boost, this is the vector $\vec{\beta} = \vec{p}/E$, such that a particle at rest with mass m is boosted to a particle with three-vector $\vec{p}$. Here, we have

$$\beta = \tanh\chi = p/E, \qquad \gamma = \cosh\chi = E/m, \qquad \beta\gamma = \sinh\chi = p/m \qquad (4.1)$$

⟨*Lorentz: public functions*⟩+≡
```
    public :: boost
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface boost
       module procedure boost_from_rest_frame
       module procedure boost_from_rest_frame_vector3
       module procedure boost_generic
       module procedure boost_canonical
    end interface
```

In the first form, the argument is some four-momentum, the space part of which determines a direction, and the associated mass (which is not checked against the four-momentum). The boost vector $\gamma\vec{\beta}$ is then given by $\vec{p}/m$. This boosts from the rest frame of a particle to the current frame. To be explicit, if $\vec{p}$ is the momentum of a particle and m its mass, $L(\vec{p}/m)$ is the transformation that turns $(m; \vec{0})$ into $(E; \vec{p})$. Conversely, the inverse transformation boosts a vector *into* the rest frame of a particle, in particular $(E; \vec{p})$ into $(m; \vec{0})$.

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function boost_from_rest_frame (p, m) result (L)
      type(lorentz_transformation_t) :: L
      type(vector4_t), intent(in) :: p
      real(default), intent(in) :: m
      L = boost_from_rest_frame_vector3 (space_part (p), m)
    end function boost_from_rest_frame
    elemental function boost_from_rest_frame_vector3 (p, m) result (L)
      type(lorentz_transformation_t) :: L
      type(vector3_t), intent(in) :: p
      real(default), intent(in) :: m
      type(vector3_t) :: beta_gamma
      real(default) :: bg2, g, c
      integer :: i,j
      if (m /= 0) then
         beta_gamma = p / m
         bg2 = beta_gamma**2
      else
         bg2 = 0
      end if
      if (bg2 /= 0) then
         g = sqrt(1 + bg2);   c = (g-1)/bg2
         L%L(0,0)  = g
         L%L(0,1:) = beta_gamma%p
         L%L(1:,0) = L%L(0,1:)
         do i=1,3
            do j=1,3
               L%L(i,j) = delta_three(i,j) + c*beta_gamma%p(i)*beta_gamma%p(j)
```

198

```
          end do
        end do
      else
        L = identity
      end if
    end function boost_from_rest_frame_vector3
```

A canonical boost is a boost along one of the coordinate axes, which we may supply as an integer argument. Here, $\gamma\beta$ is scalar.

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function boost_canonical (beta_gamma, k) result (L)
      type(lorentz_transformation_t) :: L
      real(default), intent(in) :: beta_gamma
      integer, intent(in) :: k
      real(default) :: g
      g = sqrt(1 + beta_gamma**2)
      L = identity
      L%L(0,0) = g
      L%L(0,k) = beta_gamma
      L%L(k,0) = L%L(0,k)
      L%L(k,k) = L%L(0,0)
    end function boost_canonical
```

Instead of a canonical axis, we can supply an arbitrary axis which need not be normalized. If it is zero, return the unit matrix.

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function boost_generic (beta_gamma, axis) result (L)
      type(lorentz_transformation_t) :: L
      real(default), intent(in) :: beta_gamma
      type(vector3_t), intent(in) :: axis
      if (any(axis%p/=0)) then
        L = boost_from_rest_frame_vector3 (beta_gamma * axis, axis**1)
      else
        L = identity
      end if
    end function boost_generic
```

4.1.10 Rotations

For a rotation, the vector defines the rotation axis, and its length the rotation angle.

⟨*Lorentz: public functions*⟩+≡
```
    public :: rotation
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface rotation
      module procedure rotation_generic
      module procedure rotation_canonical
      module procedure rotation_generic_cs
      module procedure rotation_canonical_cs
    end interface
```

If $\cos\phi$ and $\sin\phi$ is already known, we do not have to calculate them. Of course, the user has to ensure that $\cos^2\phi + \sin^2\phi = 1$, and that the given axis n is normalized to one. In the second form, the length of `axis` is the rotation angle.

⟨*Lorentz: subroutines*⟩+≡

```
elemental function rotation_generic_cs (cp, sp, axis) result (R)
  type(lorentz_transformation_t) :: R
  real(default), intent(in) :: cp, sp
  type(vector3_t), intent(in) :: axis
  integer :: i,j
  R = identity
  do i=1,3
     do j=1,3
        R%L(i,j) = cp*delta_three(i,j) + (1-cp)*axis%p(i)*axis%p(j)  &
             &   - sp*dot_product(epsilon_three(i,j,:), axis%p)
     end do
  end do
end function rotation_generic_cs
elemental function rotation_generic (axis) result (R)
  type(lorentz_transformation_t) :: R
  type(vector3_t), intent(in) :: axis
  real(default) :: phi
  if (any(axis%p/=0)) then
     phi = abs(axis**1)
     R = rotation_generic_cs (cos(phi), sin(phi), axis/phi)
  else
     R = identity
  end if
end function rotation_generic
```

Alternatively, give just the angle and label the coordinate axis by an integer.

⟨*Lorentz: subroutines*⟩+≡

```
elemental function rotation_canonical_cs (cp, sp, k) result (R)
  type(lorentz_transformation_t) :: R
  real(default), intent(in) :: cp, sp
  integer, intent(in) :: k
  integer :: i,j
  R = identity
  do i=1,3
     do j=1,3
        R%L(i,j) = -sp*epsilon_three(i,j,k)
     end do
     R%L(i,i) = cp
  end do
  R%L(k,k) = 1
end function rotation_canonical_cs
elemental function rotation_canonical (phi, k) result (R)
  type(lorentz_transformation_t) :: R
  real(default), intent(in) :: phi
  integer, intent(in) :: k
  R = rotation_canonical_cs(cos(phi), sin(phi), k)
end function rotation_canonical
```

This is viewed as a method for the first argument (three-vector): Reconstruct the rotation that rotates it into the second three-vector.

⟨*Lorentz: public functions*⟩+≡
```
    public :: rotation_to_2nd
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface rotation_to_2nd
       module procedure rotation_to_2nd_generic
       module procedure rotation_to_2nd_canonical
    end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function rotation_to_2nd_generic (p, q) result (R)
      type(lorentz_transformation_t) :: R
      type(vector3_t), intent(in) :: p, q
      type(vector3_t) :: a, b, ab
      real(default) :: ct, st
      if (any (p%p /= 0) .and. any (q%p /= 0)) then
         a = direction (p)
         b = direction (q)
         ab = cross_product(a,b)
         ct = a*b;   st = ab**1
         if (st /= 0) then
            R = rotation_generic_cs (ct, st, ab/st)
         else if (ct < 0) then
            R = space_reflection
         else
            R = identity
         end if
      else
         R = identity
      end if
    end function rotation_to_2nd_generic
```

The same for a canonical axis: The function returns the transformation that
rotates the k-axis into the direction of p.

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function rotation_to_2nd_canonical (k, p) result (R)
      type(lorentz_transformation_t) :: R
      integer, intent(in) :: k
      type(vector3_t), intent(in) :: p
      type(vector3_t) :: b, ab
      real(default) :: ct, st
      integer :: i, j
      if (any (p%p /= 0)) then
         b = direction (p)
         ab%p = 0
         do i = 1, 3
            do j = 1, 3
               ab%p(j) = ab%p(j) + b%p(i) * epsilon_three(i,j,k)
            end do
         end do
         ct = b%p(k);   st = ab**1
         if (st /= 0) then
            R = rotation_generic_cs (ct, st, ab/st)
         else if (ct < 0) then
            R = space_reflection
         else
```

```
          R = identity
       end if
    else
       R = identity
    end if
 end function rotation_to_2nd_canonical
```

4.1.11 Composite Lorentz transformations

This function returns the transformation that, given a pair of vectors $p_{1,2}$, (a) boosts from the rest frame of the c.m. system (with invariant mass m) into the lab frame where p_i are defined, and (b) turns the given axis (or the canonical vectors $\pm e_k$) in the rest frame into the directions of $p_{1,2}$ in the lab frame. Note that the energy components are not used; for a consistent result one should have $(p_1 + p_2)^2 = m^2$.

⟨*Lorentz: public functions*⟩+≡
```
 public :: transformation
```

⟨*Lorentz: interfaces*⟩+≡
```
 interface transformation
    module procedure transformation_rec_generic
    module procedure transformation_rec_canonical
 end interface
```

⟨*Lorentz: subroutines*⟩+≡
```
 elemental function transformation_rec_generic (axis, p1, p2, m) result (L)
   type(vector3_t), intent(in) :: axis
   type(vector4_t), intent(in) :: p1, p2
   real(default), intent(in) :: m
   type(lorentz_transformation_t) :: L
   L = boost (p1 + p2, m)
   L = L * rotation_to_2nd (axis, space_part (inverse (L) * p1))
 end function transformation_rec_generic
 elemental function transformation_rec_canonical (k, p1, p2, m) result (L)
   integer, intent(in) :: k
   type(vector4_t), intent(in) :: p1, p2
   real(default), intent(in) :: m
   type(lorentz_transformation_t) :: L
   L = boost (p1 + p2, m)
   L = L * rotation_to_2nd (k, space_part (inverse (L) * p1))
 end function transformation_rec_canonical
```

4.1.12 Applying Lorentz transformations

Multiplying vectors and Lorentz transformations is straightforward.

⟨*Lorentz: interfaces*⟩+≡
```
 interface operator(*)
    module procedure prod_LT_vector4
    module procedure prod_LT_LT
    module procedure prod_vector4_LT
 end interface
```

⟨*Lorentz: subroutines*⟩+≡
```fortran
    elemental function prod_LT_vector4 (L, p) result (np)
      type(vector4_t) :: np
      type(lorentz_transformation_t), intent(in) :: L
      type(vector4_t), intent(in) :: p
      np%p = matmul (L%L, p%p)
    end function prod_LT_vector4
    elemental function prod_LT_LT (L1, L2) result (NL)
      type(lorentz_transformation_t) :: NL
      type(lorentz_transformation_t), intent(in) :: L1,L2
      NL%L = matmul (L1%L, L2%L)
    end function prod_LT_LT
    elemental function prod_vector4_LT (p, L) result (np)
      type(vector4_t) :: np
      type(vector4_t), intent(in) :: p
      type(lorentz_transformation_t), intent(in) :: L
      np%p = matmul (p%p, L%L)
    end function prod_vector4_LT
```

4.1.13   Special Lorentz transformations

These routines have their application in the generation and extraction of angles in the phase-space sampling routine. Since this part of the program is time-critical, we calculate the composition of transformations directly instead of multiplying rotations and boosts.

This Lorentz transformation is the composition of a rotation by ϕ around the 3 axis, a rotation by θ around the 2 axis, and a boost along the 3 axis:

$$L = B_3(\beta\gamma) \, R_2(\theta) \, R_3(\phi) \tag{4.2}$$

Instead of the angles we provide sine and cosine.

⟨*Lorentz: public functions*⟩+≡
```fortran
    public :: LT_compose_r3_r2_b3
```

⟨*Lorentz: subroutines*⟩+≡
```fortran
    elemental function LT_compose_r3_r2_b3 &
        (cp, sp, ct, st, beta_gamma) result (L)
      type(lorentz_transformation_t) :: L
      real(default), intent(in) :: cp, sp, ct, st, beta_gamma
      real(default) :: gamma
      if (beta_gamma==0) then
         L%L(0,0)  = 1
         L%L(1:,0) = 0
         L%L(0,1:) = 0
         L%L(1,1:) = (/  ct*cp, -ct*sp, st /)
         L%L(2,1:) = (/     sp,     cp,  0._default /)
         L%L(3,1:) = (/ -st*cp,  st*sp, ct /)
      else
         gamma = sqrt(1 + beta_gamma**2)
         L%L(0,0)  = gamma
         L%L(1,0)  = 0
         L%L(2,0)  = 0
         L%L(3,0)  = beta_gamma
```

```
        L%L(0,1:) = beta_gamma * (/ -st*cp,  st*sp, ct /)
        L%L(1,1:) =               (/  ct*cp, -ct*sp, st /)
        L%L(2,1:) =               (/     sp,     cp, 0._default /)
        L%L(3,1:) = gamma     * (/ -st*cp,  st*sp, ct /)
      end if
    end function LT_compose_r3_r2_b3
```

Different ordering:

$$L = B_3(\beta\gamma)\, R_3(\phi)\, R_2(\theta) \tag{4.3}$$

⟨*Lorentz: public functions*⟩+≡
```
    public :: LT_compose_r2_r3_b3
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function LT_compose_r2_r3_b3 &
        (ct, st, cp, sp, beta_gamma) result (L)
      type(lorentz_transformation_t) :: L
      real(default), intent(in) :: ct, st, cp, sp, beta_gamma
      real(default) :: gamma
      if (beta_gamma==0) then
         L%L(0,0)  = 1
         L%L(1:,0) = 0
         L%L(0,1:) = 0
         L%L(1,1:) = (/  ct*cp,    -sp,      st*cp /)
         L%L(2,1:) = (/  ct*sp,     cp,      st*sp /)
         L%L(3,1:) = (/ -st   , 0._default, ct    /)
      else
         gamma = sqrt(1 + beta_gamma**2)
         L%L(0,0)  = gamma
         L%L(1,0)  = 0
         L%L(2,0)  = 0
         L%L(3,0)  = beta_gamma
         L%L(0,1:) = beta_gamma * (/ -st   , 0._default, ct    /)
         L%L(1,1:) =              (/  ct*cp,    -sp,      st*cp /)
         L%L(2,1:) =              (/  ct*sp,     cp,      st*sp /)
         L%L(3,1:) = gamma     * (/ -st   , 0._default, ct    /)
      end if
    end function LT_compose_r2_r3_b3
```

This function returns the previous Lorentz transformation applied to an arbitrary four-momentum and extracts the space part of the result:

$$\vec{n} = [B_3(\beta\gamma)\, R_2(\theta)\, R_3(\phi)\, p]_{\text{space part}} \tag{4.4}$$

The second variant applies if there is no rotation

⟨*Lorentz: public functions*⟩+≡
```
    public :: axis_from_p_r3_r2_b3, axis_from_p_b3
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function axis_from_p_r3_r2_b3 &
        (p, cp, sp, ct, st, beta_gamma) result (n)
      type(vector3_t) :: n
      type(vector4_t), intent(in) :: p
      real(default), intent(in) :: cp, sp, ct, st, beta_gamma
```

```
      real(default) :: gamma, px, py
      px = cp * p%p(1) - sp * p%p(2)
      py = sp * p%p(1) + cp * p%p(2)
      n%p(1) =  ct * px + st * p%p(3)
      n%p(2) = py
      n%p(3) = -st * px + ct * p%p(3)
      if (beta_gamma/=0) then
         gamma = sqrt(1 + beta_gamma**2)
         n%p(3) = n%p(3) * gamma + p%p(0) * beta_gamma
      end if
    end function axis_from_p_r3_r2_b3

    elemental function axis_from_p_b3 (p, beta_gamma) result (n)
      type(vector3_t) :: n
      type(vector4_t), intent(in) :: p
      real(default), intent(in) :: beta_gamma
      real(default) :: gamma
      n%p = p%p(1:3)
      if (beta_gamma/=0) then
         gamma = sqrt(1 + beta_gamma**2)
         n%p(3) = n%p(3) * gamma + p%p(0) * beta_gamma
      end if
    end function axis_from_p_b3
```

4.1.14 Special functions

The standard phase space function:

⟨*Lorentz: public functions*⟩+≡
```
    public :: lambda
```

⟨*Lorentz: subroutines*⟩+≡
```
    elemental function lambda (m1sq, m2sq, m3sq)
      real(default) :: lambda
      real(default), intent(in) :: m1sq, m2sq, m3sq
      lambda = (m1sq - m2sq - m3sq)**2 - 4*m2sq*m3sq
    end function lambda
```

Return a pair of head-to-head colliding momenta, given the collider energy, particle masses, and optionally the momentum of the c.m. system.

⟨*Lorentz: public functions*⟩+≡
```
    public :: colliding_momenta
```

⟨*Lorentz: subroutines*⟩+≡
```
    function colliding_momenta (sqrts, m, p_cm) result (p)
      type(vector4_t), dimension(2) :: p
      real(default), intent(in) :: sqrts
      real(default), dimension(2), intent(in), optional :: m
      real(default), intent(in), optional :: p_cm
      real(default), dimension(2) :: dmsq
      real(default) :: ch, sh
      real(default), dimension(2) :: E0, p0
      integer, dimension(2), parameter :: sgn = (/1, -1/)
      if (sqrts == 0) then
```

```
         call msg_fatal (" Colliding beams: sqrts is zero (please set sqrts)")
         p = vector4_null;  return
      else if (sqrts <= 0) then
         call msg_fatal (" Colliding beams: sqrts is negative")
         p = vector4_null;  return
      end if
      if (present (m)) then
         dmsq = sgn * (m(1)**2-m(2)**2)
         E0 = (sqrts + dmsq/sqrts) / 2
         if (any (E0 < m)) then
            call msg_fatal &
                 (" Colliding beams: beam energy is less than particle mass")
            p = vector4_null;  return
         end if
         p0 = sgn * sqrt (E0**2 - m**2)
      else
         E0 = sqrts / 2
         p0 = sgn * E0
      end if
      if (present (p_cm)) then
         sh = p_cm / sqrts
         ch = sqrt (1 + sh**2)
         p = vector4_moving (E0 * ch + p0 * sh, E0 * sh + p0 * ch, 3)
      else
         p = vector4_moving (E0, p0, 3)
      end if
    end function colliding_momenta
```

4.2 Special Physics functions

Define special kinematical functions like running α_s and e.g. Catani-Seymour dipole terms.

To make maximum use of this, all functions, if possible, are declared elemental (or pure, if this is not possible).

⟨sm_physics.f90⟩≡
  ⟨*File header*⟩

  module sm_physics

  ⟨*Use kinds*⟩
    use constants !NODEP!
  ⟨*Use file utils*⟩
    use diagnostics !NODEP!
    use lorentz !NODEP!

  ⟨*Standard module head*⟩

  ⟨*SM physics: public parameters*⟩

  ⟨*SM physics: public functions*⟩

  contains

  ⟨*SM physics: subroutines*⟩
  end module sm_physics

First we set a reference value for $\alpha_s(M_Z) = 0.1178$.

⟨*SM physics: public parameters*⟩≡
    real(kind=default), public, parameter :: as_mz = 0.1178_default, &
      mass_z = 91.188_default

Then we define the coefficients of the beta function of QCD (as a reference cf. the Particle Data Group), where n_f is the number of active flavors in two different schemes:

$$\beta_0 = 11 - \frac{2}{3}n_f \tag{4.5}$$

$$\beta_1 = 51 - \frac{19}{3}n_f \tag{4.6}$$

$$\beta_2 = 2857 - \frac{5033}{9}n_f + \frac{325}{27}n_f^2 \tag{4.7}$$

$$b_0 = \frac{1}{12\pi}\left(11C_A - 2n_f\right) \tag{4.8}$$

$$b_1 = \frac{1}{24\pi^2}\left(17C_A^2 - 5C_A n_f - 3C_F n_f\right) \tag{4.9}$$

$$b_2 = \frac{1}{(4\pi)^3}\left(\frac{2857}{54}C_A^3 - \frac{1415}{54} * C_A^2 n_f - \frac{205}{18}C_A C_F n_f + C_F^2 n_f + \frac{79}{54}C_A n_f **2 + \frac{11}{9}C_F n_f **2\right) \tag{4.10}$$

⟨*SM physics: public functions*⟩≡
    public :: beta0, beta1, beta2, coeff_b0, coeff_b1, coeff_b2

⟨*SM physics: subroutines*⟩≡

```
pure function beta0 (nf)
  real(kind=default), intent(in) :: nf
  real(kind=default) :: beta0
  beta0 = 11.0_default - two/three * nf
end function beta0

pure function beta1 (nf)
  real(kind=default), intent(in) :: nf
  real(kind=default) :: beta1
  beta1 = 51.0_default - 19.0_default/three * nf
end function beta1

pure function beta2 (nf)
  real(kind=default), intent(in) :: nf
  real(kind=default) :: beta2
  beta2 = 2857.0_default - 5033.0_default / 9.0_default * &
                 nf + 325.0_default/27.0_default * nf**2
end function beta2

pure function coeff_b0 (nf)
  real(kind=default), intent(in) :: nf
  real(kind=default) :: coeff_b0
  coeff_b0 = (11.0_default * CA - two * nf) / (12.0_default * pi)
end function coeff_b0

pure function coeff_b1 (nf)
  real(kind=default), intent(in) :: nf
  real(kind=default) :: coeff_b1
  coeff_b1 = (17.0_default * CA**2 - five * CA * nf - three * CF * nf) / &
            (24.0_default * pi**2)
end function coeff_b1

pure function coeff_b2 (nf)
  real(kind=default), intent(in) :: nf
  real(kind=default) :: coeff_b2
  coeff_b2 = (2857.0_default/54.0_default * CA**3 - &
                 1415.0_default/54.0_default * &
                 CA**2 * nf - 205.0_default/18.0_default * CA*CF*nf &
                 + 79.0_default/54.0_default * CA*nf**2 + &
                 11.0_default/9.0_default * CF * nf**2) / (four*pi)**3
end function coeff_b2
```

There should be two versions of running α_s, one which takes the scale and Λ_{QCD} as imput, and one which takes the scale and e.g. $\alpha_s(m_Z)$ as input. Here, we take the one which takes the QCD scale and scale as inputs from the PDG book.

⟨*SM physics: public functions*⟩+≡

```
public :: running_as, running_as_lam
```

⟨*SM physics: subroutines*⟩+≡

```
pure function running_as (scale,al_mz,mz,order,nf) result (ascale)
  real(kind=default), intent(in) :: scale
  real(kind=default), intent(in), optional :: al_mz, nf, mz
  integer, intent(in), optional :: order
```

```
    integer :: ord
    real(kind=default) :: az, m_z, as_log, n_f, b0, b1, b2, ascale
    real(kind=default) :: as0, as1
    if (present(mz)) then
      m_z = mz
    else
      m_z = mass_z
    end if
    if (present(order)) then
      ord = order
    else
      ord = 0
    end if
    if (present(al_mz)) then
      az = al_mz
    else
      az = as_mz
    end if
    if (present(nf)) then
      n_f = nf
    else
      n_f = 5
    end if
    b0 = coeff_b0 (n_f)
    b1 = coeff_b1 (n_f)
    b2 = coeff_b2 (n_f)
    as_log = one + b0 * az * log(scale**2/m_z**2)
    as0 = az / as_log
    as1 = as0 - as0**2 * b1/b0 * log(as_log)
    select case (ord)
      case (0)
        ascale = as0
      case (1)
        ascale = as1
      case (2)
        ascale = as1 + as0**3 * (b1**2/b0**2 * ((log(as_log))**2 - &
                 log(as_log) + as_log - one) - b2/b0 * (as_log - one))
    end select
end function running_as

pure function running_as_lam (nf,scale,lambda_qcd,order) result (ascale)
  real(kind=default), intent(in) :: nf, scale, lambda_qcd
  integer, intent(in), optional :: order
  real(kind=default) :: as0, as1, logmul, b0, b1, b2, ascale
  integer :: ord
  if (present(order)) then
    ord = order
  else
    ord = 0
  end if
  b0 = beta0(nf)
  b1 = beta1(nf)
  b2 = beta2(nf)
  logmul = log(scale**2/lambda_qcd**2)
```

```
       as0 = four*pi / b0 / logmul
       as1 = as0 * (one - two* b1 / b0**2 * log(logmul) / logmul)
       select case (ord)
         case (0)
           ascale = as0
         case (1)
           ascale = as1
         case (2)
           ascale = as1 + as0 * four * b1**2/b0**4/logmul**2 * &
                 ( (log(logmul) - 0.5_default)**2 + &
                 b2*b0/8.0_default/b1**2 - five/four)
       end select
     end function running_as_lam
```

These are fundamental constants of the Catani-Seymour dipole formalism. Since the corresponding parameters for the gluon case depend on the number of flavors which is treated as an argument, there we do have functions and not parameters.

$$\gamma_q = \gamma_{\bar{q}} = \frac{3}{2}C_F \qquad \gamma_g = \frac{11}{6}C_A - \frac{2}{3}T_R N_f \qquad (4.11)$$

$$K_q = K_{\bar{q}} = \left(\frac{7}{2} - \frac{\pi^2}{6}\right)C_F \qquad K_g = \left(\frac{67}{18} - \frac{\pi^2}{6}\right)C_A - \frac{10}{9}T_R N_f \qquad (4.12)$$

⟨*SM physics: public parameters*⟩+≡
```
     real(kind=default), parameter, public ::  gamma_q = three/two * CF, &
       k_q = (7.0_default/two - pi**2/6.0_default) * CF
```

⟨*SM physics: public functions*⟩+≡
```
     public :: gamma_g, k_g
```
⟨*SM physics: subroutines*⟩+≡
```
     elemental function gamma_g (nf) result (gg)
       real(kind=default), intent(in) :: nf
       real(kind=default) :: gg
       gg = 11.0_default/6.0_default * CA - two/three * TR * nf
     end function gamma_g

     elemental function k_g (nf) result (kg)
       real(kind=default), intent(in) :: nf
       real(kind=default) :: kg
       kg = (67.0_default/18.0_default - pi**2/6.0_default) * CA - &
                 10.0_default/9.0_default * TR * nf
     end function k_g
```

The dilogarithm. This simplified version is bound to double precision, and restricted to argument values less or equal to unity, so we do not need complex algebra. The wrapper converts it to default precision (which is, of course, a no-op if double=default).

The routine calculates the dilogarithm through mapping on the area where there is a quickly convergent series (adapted from an F77 routine by Hans Kuijf, 1988): Map x such that x is not in the neighbourhood of 1. Note that $|z| = -\ln(1-x)$ is always smaller than 1.10, but $\frac{1 \cdot 10^{19}}{19!}$Bernoulli$_{19} = 2.7 \times 10^{-15}$.
⟨*SM physics: public functions*⟩+≡

```
        public :: Li2
```

⟨SM physics: subroutines⟩+≡
```
    function Li2 (x)
        use kinds, only: double !NODEP!
        real(default), intent(in) :: x
        real(default) :: Li2
        Li2 = real( Li2_double (real(x, kind=double)), kind=default)
    end function Li2
```

⟨SM physics: subroutines⟩+≡
```
    function Li2_double (x)  result (Li2)
      use kinds, only: double !NODEP!
      real(kind=double), intent(in) :: x
      real(kind=double) :: Li2
      real(kind=double), parameter :: pi2_6 = pi**2/6
      if (abs(1-x) < 1.E-13_double) then
         Li2 = pi2_6
      else if (abs(1-x) <  0.5_double) then
         Li2 = pi2_6 - log(1-x) * log(x) - Li2_restricted (1-x)
      else if (abs(x).gt.1.d0) then
         call msg_bug (" Dilogarithm called outside of defined range.")
  !       Li2 = -pi2_6 - 0.5_default * log(-x) * log(-x) - Li2_restricted (1/x)
      else
         Li2 = Li2_restricted (x)
      end if
    contains
      function Li2_restricted (x) result (Li2)
        real(kind=double), intent(in) :: x
        real(kind=double) :: Li2
        real(kind=double) :: tmp, z, z2
        z = - log (1-x)
        z2 = z**2
  ! Horner's rule for the powers z^3 through z^19
        tmp = 43867._double/798._double
        tmp = tmp * z2 /342._double - 3617._double/510._double
        tmp = tmp * z2 /272._double + 7._double/6._double
        tmp = tmp * z2 /210._double - 691._double/2730._double
        tmp = tmp * z2 /156._double + 5._double/66._double
        tmp = tmp * z2 /110._double - 1._double/30._double
        tmp = tmp * z2 / 72._double + 1._double/42._double
        tmp = tmp * z2 / 42._double - 1._double/30._double
        tmp = tmp * z2 / 20._double + 1._double/6._double
  ! The first three terms of the power series
        Li2 = z2 * z * tmp / 6._double - 0.25_double * z2 + z
      end function Li2_restricted
    end function Li2_double
```

⟨SM physics: public functions⟩+≡
```
  public :: faux
```

⟨SM physics: subroutines⟩+≡
```
    elemental function faux (x) result (y)
      real(default), intent(in) :: x
```

```
      complex(default) :: y
      if (1 <= x) then
         y = asin(sqrt(1/x))**2
      else
         y = - 1/4.0_default * (log((1 + sqrt(1 - x))/ &
               (1 - sqrt(1 - x))) - cmplx (0.0_default, pi, kind=default))**2
      end if
   end function faux
```

⟨*SM physics: public functions*⟩+≡
```
      public :: fonehalf
```

⟨*SM physics: subroutines*⟩+≡
```
   elemental function fonehalf (x) result (y)
      real(default), intent(in) :: x
      complex(default) :: y
      if (x==0) then
         y = 0
      else
         y = - 2.0_default * x * (1 + (1 - x) * faux(x))
      end if
   end function fonehalf
```

⟨*SM physics: public functions*⟩+≡
```
      public :: fonehalf_pseudo
```

⟨*SM physics: subroutines*⟩+≡
```
   function fonehalf_pseudo (x) result (y)
      real(default), intent(in) :: x
      complex(default) :: y
      if (x==0) then
         y = 0
      else
         y = - 2.0_default * x * faux(x)
      end if
   end function fonehalf_pseudo
```

⟨*SM physics: public functions*⟩+≡
```
      public :: fone
```

⟨*SM physics: subroutines*⟩+≡
```
   elemental function fone (x) result  (y)
      real(default), intent(in) :: x
      complex(default) :: y
      if (x==0) then
         y = 2.0_default
      else
         y = 2.0_default + 3.0_default * x + &
               3.0_default * x * (2.0_default - x) * &
               faux(x)
      end if
   end function fone
```

⟨*SM physics: public functions*⟩+≡
```
    public :: gaux
```

⟨*SM physics: subroutines*⟩+≡
```
    elemental function gaux (x) result (y)
      real(default), intent(in) :: x
      complex(default) :: y
      if (1 <= x) then
         y = sqrt(x - 1) * asin(sqrt(1/x))
      else
         y = sqrt(1 - x) * (log((1 + sqrt(1 - x)) / &
               (1 - sqrt(1 - x))) - cmplx (0.0_default, pi, kind=default)) / 2
      end if
    end function gaux
```


⟨*SM physics: public functions*⟩+≡
```
    public :: tri_i1
```

⟨*SM physics: subroutines*⟩+≡
```
    elemental function tri_i1 (a,b) result (y)
      real(default), intent(in) :: a,b
      complex(default) :: y
      y = a*b/2.0_default/(a-b) + a**2 * b**2/2.0_default/(a-b)**2 * &
            (faux(a) - faux(b)) + &
            a**2 * b/(a-b)**2 * (gaux(a) - gaux(b))
    end function tri_i1
```


⟨*SM physics: public functions*⟩+≡
```
    public :: tri_i2
```

⟨*SM physics: subroutines*⟩+≡
```
    elemental function tri_i2 (a,b) result (y)
      real(default), intent(in) :: a,b
      complex(default) :: y
      y = - a * b / 2.0_default / (a-b) * (faux(a) - faux(b))
    end function tri_i2
```


These functions are for the running of the strong coupling constants, α_s.

⟨*SM physics: public functions*⟩+≡
```
    public :: run_b0
```

⟨*SM physics: subroutines*⟩+≡
```
    elemental function run_b0 (nf) result (bnull)
      integer, intent(in) :: nf
      real(default) :: bnull
      bnull = 33.0_default - 2.0_default * nf
    end function run_b0
```


⟨*SM physics: public functions*⟩+≡
```
    public :: run_b1
```

⟨*SM physics: subroutines*⟩+≡
```
    elemental function run_b1 (nf) result (bone)
        integer, intent(in) :: nf
        real(default) :: bone
        bone = 6.0_default * (153.0_default - 19.0_default * nf)/run_b0(nf)**2
    end function run_b1
```

⟨*SM physics: public functions*⟩+≡
```
    public :: run_aa
```

⟨*SM physics: subroutines*⟩+≡
```
    elemental function run_aa (nf) result (aaa)
        integer, intent(in) :: nf
        real(default) :: aaa
        aaa = 12.0_default * PI / run_b0(nf)
    end function run_aa
```

⟨*SM physics: pubic functions*⟩≡
```
    public :: run_bb
```

⟨*SM physics: subroutines*⟩+≡
```
    elemental function run_bb (nf) result (bbb)
      integer, intent(in) :: nf
      real(default) :: bbb
      bbb = run_b1(nf) / run_aa(nf)
    end function run_bb
```

4.2.1 Functions for Catani-Seymour dipoles

For the automated Catani-Seymour dipole subtraction, we need the following functions.

⟨*SM physics: public functions*⟩+≡
```
    public :: ff_dipole
```

⟨*SM physics: subroutines*⟩+≡
```
    pure subroutine ff_dipole (v_ijk,y_ijk,p_ij,pp_k,p_i,p_j,p_k)
        type(vector4_t), intent(in) :: p_i, p_j, p_k
        type(vector4_t), intent(out) :: p_ij, pp_k
        real(kind=default), intent(out) :: y_ijk
        real(kind=default) :: z_i
        real(kind=default), intent(out) :: v_ijk
        z_i   = (p_i*p_k) / ((p_k*p_j) + (p_k*p_i))
        y_ijk = (p_i*p_j) / ((p_i*p_j) + (p_i*p_k) + (p_j*p_k))
        p_ij  = p_i + p_j - y_ijk/(1.0_default - y_ijk) * p_k
        pp_k  = (1.0/(1.0_default - y_ijk)) * p_k
        !!! We don't multiply by alpha_s right here:
        v_ijk = 8.0_default * PI * CF * &
            (2.0 / (1.0 - z_i*(1.0 - y_ijk)) - (1.0 + z_i))
    end subroutine ff_dipole
```

⟨*SM physics: public functions*⟩+≡
```
    public :: fi_dipole
```

⟨*SM physics: subroutines*⟩+≡
```
    pure subroutine fi_dipole (v_ija,x_ija,p_ij,pp_a,p_i,p_j,p_a)
        type(vector4_t), intent(in) :: p_i, p_j, p_a
        type(vector4_t), intent(out) :: p_ij, pp_a
        real(kind=default), intent(out) :: x_ija
        real(kind=default) :: z_i
        real(kind=default), intent(out) :: v_ija
        z_i  = (p_i*p_a) / ((p_a*p_j) + (p_a*p_i))
        x_ija = ((p_i*p_a) + (p_j*p_a) - (p_i*p_j)) &
             / ((p_i*p_a) + (p_j*p_a))
        p_ij  = p_i + p_j - (1.0_default - x_ija) * p_a
        pp_a  = x_ija * p_a
        !!! We don't not multiply by alpha_s right here:
        v_ija = 8.0_default * PI * CF * &
             (2.0 / (1.0 - z_i + (1.0 - x_ija)) - (1.0 + z_i)) / x_ija
    end subroutine fi_dipole
```

⟨*SM physics: public functions*⟩+≡
```
    public :: if_dipole
```

⟨*SM physics: subroutines*⟩+≡
```
    pure subroutine if_dipole (v_kja,u_j,p_aj,pp_k,p_k,p_j,p_a)
        type(vector4_t), intent(in) :: p_k, p_j, p_a
        type(vector4_t), intent(out) :: p_aj, pp_k
        real(kind=default), intent(out) :: u_j
        real(kind=default) :: x_kja
        real(kind=default), intent(out) :: v_kja
        u_j   = (p_a*p_j) / ((p_a*p_j) + (p_a*p_k))
        x_kja = ((p_a*p_k) + (p_a*p_j) - (p_j*p_k)) &
             / ((p_a*p_j) + (p_a*p_k))
        p_aj  = x_kja * p_a
        pp_k  = p_k + p_j - (1.0_default - x_kja) * p_a
        v_kja = 8.0_default * PI * CF * &
             (2.0 / (1.0 - x_kja + u_j) - (1.0 + x_kja)) / x_kja
    end subroutine if_dipole
```

This function depends on a variable number of final state particles whose kinematics all get changed by the initial-initial dipole insertion.

⟨*SM physics: public functions*⟩+≡
```
    public :: ii_dipole
```

⟨*SM physics: subroutines*⟩+≡
```
    pure subroutine ii_dipole (v_jab,v_j,p_in,p_out,flag_1or2)
        type(vector4_t), dimension(:), intent(in) :: p_in
        type(vector4_t), dimension(size(p_in)-1), intent(out) :: p_out
        logical, intent(in) :: flag_1or2
        real(kind=default), intent(out) :: v_j
        real(kind=default), intent(out) :: v_jab
        type(vector4_t) :: p_a, p_b, p_j
        type(vector4_t) :: k, kk
        type(vector4_t) :: p_aj
        real(kind=default) :: x_jab
        integer :: i
        !!! flag_1or2 decides whether this a 12 or 21 dipole
```

```
        if (flag_1or2) then
           p_a = p_in(1)
           p_b = p_in(2)
        else
           p_b = p_in(1)
           p_a = p_in(2)
        end if
        !!! We assume that the unresolved particle has always the last
        !!! momentum
        p_j = p_in(size(p_in))
        x_jab = ((p_a*p_b) - (p_a*p_j) - (p_b*p_j)) / (p_a*p_b)
        v_j = (p_a*p_j) / (p_a * p_b)
        p_aj  = x_jab * p_a
        k     = p_a + p_b - p_j
        kk    = p_aj + p_b
        do i = 3, size(p_in)-1
           p_out(i) = p_in(i) - 2.0*((k+kk)*p_in(i))/((k+kk)*(k+kk)) * (k+kk) + &
                 (2.0 * (k*p_in(i)) / (k*k)) * kk
        end do
        if (flag_1or2) then
           p_out(1) = p_aj
           p_out(2) = p_b
        else
           p_out(1) = p_b
           p_out(2) = p_aj
        end if
        v_jab = 8.0_default * PI * CF * &
              (2.0 / (1.0 - x_jab) - (1.0 + x_jab)) / x_jab
   end subroutine ii_dipole
```

4.2.2 Distributions for integrated dipoles and such

The Dirac delta distribution, modified for Monte-Carlo sampling:

⟨*SM physics: public functions*⟩+≡
```
    public :: delta
```

⟨*SM physics: subroutines*⟩+≡
```
    elemental function delta (x,eps) result (z)
       real(kind=default), intent(in) :: x, eps
       real(kind=default) :: z
       if (x > 1.0_default - eps) then
          z = 1.0_default/eps
       else
          z = 0
       end if
    end function delta
```

The +-distribution, $P_+(x) = \left(\frac{1}{1-x}\right)_+$, for the regularization of soft-collinear singularities. The constant part for the Monte-Carlo sampling is the integral over the splitting function divided by the weight for the WHIZARD numerical integration over the interval.

⟨*SM physics: public functions*⟩+≡
```
    public :: plus_distr
```

⟨*SM physics: subroutines*⟩+≡
```
    elemental function plus_distr (x,eps) result (plusd)
       real(kind=default), intent(in) :: x, eps
       real(kind=default) :: plusd
       if (x > (1.0_default - eps)) then
          plusd = log(eps)/eps
       else
          plusd = one/(one-x)
       end if
    end function plus_distr
```

The splitting function in $D = 4$ dimensions, regularized as +-distributions if necessary:

$$P^{qq}(x) = P^{\bar{q}\bar{q}}(x) = C_F \cdot \left(\frac{1 + x^2}{1 - x}\right)_+ \tag{4.13}$$

$$P^{qg}(x) = P^{\bar{q}g}(x) = C_F \cdot \frac{1 + (1 - x)^2}{x} \tag{4.14}$$

$$P^{gq}(x) = P^{g\bar{q}}(x) = T_R \cdot \left[x^2 + (1 - x)^2\right] \tag{4.15}$$

$$P^{gg}(x) = 2C_A\left[\left(\frac{1}{1 - x}\right)_+ + \frac{1 - x}{x} - 1 + x(1 - x)\right] + \delta(1 - x)\left(\frac{11}{6}C_A - \frac{2}{3}N_f T_R\right) \tag{4.16}$$

Since the number of flavors summed over in the gluon splitting function might depend on the physics case under consideration it is implemented as an input variable.

⟨*SM physics: public functions*⟩+≡
```
    public :: pqq
```

⟨*SM physics: subroutines*⟩+≡
```
    elemental function pqq (x,eps) result (pqqx)
       real(kind=default), intent(in) :: x, eps
       real(kind=default) :: pqqx
       if (x > (1.0_default - eps)) then
          pqqx = (eps-one)/two + two*log(eps)/eps - three*(eps-one)/eps/two
       else
          pqqx = (one + x**2)/(one-x)
       end if
       pqqx = CF * pqqx
    end function pqq
```

⟨*SM physics: public functions*⟩+≡
```
    public :: pgq
```

⟨*SM physics: subroutines*⟩+≡
```
    elemental function pgq (x) result (pgqx)
       real(kind=default), intent(in) :: x
       real(kind=default) :: pgqx
       pgqx = TR * (x**2 + (one - x)**2)
    end function pgq
```

⟨*SM physics: public functions*⟩+≡
```
    public :: pqg
```

⟨*SM physics: subroutines*⟩+≡
```
    elemental function pqg (x) result (pqgx)
      real(kind=default), intent(in) :: x
      real(kind=default) :: pqgx
      pqgx = CF * (one + (one - x)**2) / x
    end function pqg
```

⟨*SM physics: public functions*⟩+≡
```
    public :: pgg
```

⟨*SM physics: subroutines*⟩+≡
```
    elemental function pgg (x, nf, eps) result (pggx)
      real(kind=default), intent(in) :: x, nf, eps
      real(kind=default) :: pggx
      pggx = two * CA * ( plus_distr (x, eps) + (one-x)/x - one + &
                   x*(one-x)) + delta (x, eps)  * gamma_g(nf)
    end function pgg
```

For the *qq* and *gg* cases, there exist "regularized" versions of the splitting functions:

$$P_{\text{reg}}^{qq}(x) = -C_F \cdot (1 + x) \tag{4.17}$$

$$P_{\text{reg}}^{gg}(x) = 2C_A \left[\frac{1-x}{x} - 1 + x(1-x) \right] \tag{4.18}$$

$$\tag{4.19}$$

⟨*SM physics: public functions*⟩+≡
```
    public :: pqq_reg
```

⟨*SM physics: subroutines*⟩+≡
```
    elemental function pqq_reg (x) result (pqqregx)
      real(kind=default), intent(in) :: x
      real(kind=default) :: pqqregx
      pqqregx = - CF * (one + x)
    end function pqq_reg
```

⟨*SM physics: public functions*⟩+≡
```
    public :: pgg_reg
```

⟨*SM physics: subroutines*⟩+≡
```
    elemental function pgg_reg (x) result (pggregx)
      real(kind=default), intent(in) :: x
      real(kind=default) :: pggregx
      pggregx = two * CA * ((one - x)/x - one + x*(one - x))
    end function pgg_reg
```

Here, we collect the expressions needed for integrated Catani-Seymour dipoles, and the so-called flavor kernels. We always distinguish between the "ordinary" Catani-Seymour version, and the one including a phase-space slicing parameter, α.

The standard flavor kernels $\overline{K}^{ab}$ are:

$$\overline{K}^{qg}(x) = \overline{K}^{\bar{q}g}(x) = P^{qg}(x)\log((1-x)/x) + CF \times x \qquad (4.20)$$

$$\overline{K}^{gq}(x) = \overline{K}^{g\bar{q}}(x) = P^{gq}(x)\log((1-x)/x) + TR \times 2x(1-x) \qquad (4.21)$$

$$\overline{K}^{qq} = C_F\left[\left(\frac{2}{1-x}\log\frac{1-x}{x}\right)_+ - (1+x)\log((1-x)/x) + (1-x)\right] - (5-\pi^2)\cdot C_F \cdot \delta(1-x)$$
$$(4.22)$$

$$\overline{K}^{gg} = 2C_A\left[\left(\frac{1}{1-x}\log\frac{1-x}{x}\right)_+ + \left(\frac{1-x}{x} - 1 + x(1-x)\right)\log((1-x)/x)\right] - delta(1-x)\left[(f\right.$$
$$(4.23)$$

⟨*SM physics: public functions*⟩+≡
```
    public :: kbarqg
```

⟨*SM physics: subroutines*⟩+≡
```
    function kbarqg (x) result (kbarqgx)
      real(kind=default), intent(in) :: x
      real(kind=default) :: kbarqgx
      kbarqgx = pqg(x) * log((one-x)/x) + CF * x
    end function kbarqg
```

⟨*SM physics: public functions*⟩+≡
```
    public :: kbargq
```

⟨*SM physics: subroutines*⟩+≡
```
    function kbargq (x) result (kbargqx)
      real(kind=default), intent(in) :: x
      real(kind=default) :: kbargqx
      kbargqx = pgq(x) * log((one-x)/x) + two * TR * x * (one - x)
    end function kbargq
```

⟨*SM physics: public functions*⟩+≡
```
    public :: kbarqq
```

⟨*SM physics: subroutines*⟩+≡
```
    function kbarqq (x,eps) result (kbarqqx)
      real(kind=default), intent(in) :: x, eps
      real(kind=default) :: kbarqqx
      kbarqqx = CF*(log_plus_distr(x,eps) - (one+x) * log((one-x)/x) + (one - &
          x) - (five - pi**2) * delta(x,eps))
    end function kbarqq
```

⟨*SM physics: public functions*⟩+≡
```
    public :: kbargg
```

⟨*SM physics: subroutines*⟩+≡
```
    function kbargg (x,eps,nf) result (kbarggx)
      real(kind=default), intent(in) :: x, eps, nf
      real(kind=default) :: kbarggx
      kbarggx = CA * (log_plus_distr(x,eps) + two * ((one-x)/x - one + &
                      x*(one-x) * log((1-x)/x))) - delta(x,eps) * &
                      ((50.0_default/9.0_default - pi**2) * CA - &
```

```
                    16.0_default/9.0_default * TR * nf)
    end function kbargg
```

The $\tilde{K}$ are used when two identified hadrons participate:

$$\tilde{K}^{ab}(x) = P_{\mathrm{reg}}^{ab}(x) \cdot \log(1-x) + \delta^{ab}\mathbf{T}_a^2 \left[\left(\frac{2}{1-x} \log(1-x) \right)_+ - \frac{\pi^2}{3} \delta(1-x) \right] \quad (4.24)$$

⟨*SM physics: public functions*⟩+≡
```
    public :: ktildeqq
```

⟨*SM physics: subroutines*⟩+≡
```
    function ktildeqq (x,eps) result (ktildeqqx)
      real(kind=default), intent(in) :: x, eps
      real(kind=default) :: ktildeqqx
      ktildeqqx = pqq_reg (x) * log(one-x) + CF * ( - log2_plus_distr (x,eps) &
                          - pi**2/three * delta(x,eps))
    end function ktildeqq
```

⟨*SM physics: public functions*⟩+≡
```
    public :: ktildeqg
```

⟨*SM physics: subroutines*⟩+≡
```
    function ktildeqg (x,eps) result (ktildeqgx)
      real(kind=default), intent(in) :: x, eps
      real(kind=default) :: ktildeqgx
      ktildeqgx = pqg (x) * log(one-x)
    end function ktildeqg
```

⟨*SM physics: public functions*⟩+≡
```
    public :: ktildegq
```

⟨*SM physics: subroutines*⟩+≡
```
    function ktildegq (x,eps) result (ktildegqx)
      real(kind=default), intent(in) :: x, eps
      real(kind=default) :: ktildegqx
      ktildegqx = pgq (x) * log(one-x)
    end function ktildegq
```

⟨*SM physics: public functions*⟩+≡
```
    public :: ktildegg
```

⟨*SM physics: subroutines*⟩+≡
```
    function ktildegg (x,eps) result (ktildeggx)
      real(kind=default), intent(in) :: x, eps
      real(kind=default) :: ktildeggx
      ktildeggx = pgg_reg (x) * log(one-x) + CA * ( - &
          log2_plus_distr (x,eps) - pi**2/three * delta(x,eps))
    end function ktildegg
```

The insertion operator might not be necessary for a GOLEM interface but is demanded by the Les Houches NLO accord. It is a three-dimensional array, where the index always gives the inverse power of the DREG expansion parameter, ϵ.
⟨*SM physics: public functions*⟩+≡
```
    public :: insert_q
```

⟨*SM physics: subroutines*⟩+≡
```
    pure function insert_q ()
      real(kind=default), dimension(0:2) :: insert_q
      insert_q(0) = gamma_q + k_q - pi**2/three * CF
      insert_q(1) = gamma_q
      insert_q(2) = CF
    end function insert_q
```

⟨*SM physics: public functions*⟩+≡
```
    public :: insert_g
```

⟨*SM physics: subroutines*⟩+≡
```
    pure function insert_g (nf)
      real(kind=default), intent(in) :: nf
      real(kind=default), dimension(0:2) :: insert_g
      insert_g(0) = gamma_g (nf) + k_g (nf) - pi**2/three * CA
      insert_g(1) = gamma_g (nf)
      insert_g(2) = CA
    end function insert_g
```

For better convergence, one can exclude regions of phase space with a slicing parameter from the dipole subtraction procedure. First of all, the K functions get modified:

$$K_i(\alpha) = K_i - \mathbf{T}_i^2 \log^2 \alpha + \gamma_i(\alpha - 1 - \log \alpha) \tag{4.25}$$

⟨*SM physics: public functions*⟩+≡
```
    public :: k_q_al, k_g_al
```

⟨*SM physics: subroutines*⟩+≡
```
    pure function k_q_al (alpha)
      real(kind=default), intent(in) :: alpha
      real(kind=default) :: k_q_al
      k_q_al = k_q - CF * (log(alpha))**2 + gamma_q * &
                      (alpha - one - log(alpha))
    end function k_q_al

    pure function k_g_al (alpha, nf)
      real(kind=default), intent(in) :: alpha, nf
      real(kind=default) :: k_g_al
      k_g_al = k_g (nf) - CA * (log(alpha))**2 + gamma_g (nf) * &
                      (alpha - one - log(alpha))
    end function k_g_al
```

The +-distribution, but with a phase-space slicing parameter, α, $P_{1-alpha}(x) = \left(\frac{1}{1-x}\right)_{1-x}$. Since we need the fatal error message here, this function cannot be elemental.

⟨*SM physics: public functions*⟩+≡
```
    public :: plus_distr_al
```

⟨*SM physics: subroutines*⟩+≡
```
    function plus_distr_al (x,alpha,eps) result (plusd_al)
       real(kind=default), intent(in) :: x,  eps, alpha
       real(kind=default) :: plusd_al
```

```
      if ((1.0_default - alpha) .ge. (1.0_default - eps)) then
         call msg_fatal ('sm_physics, plus_distr_al: alpha and epsilon chosen wrongly')
      elseif (x < (1.0_default - alpha)) then
         plusd_al = 0
      else if (x > (1.0_default - eps)) then
         plusd_al = log(eps/alpha)/eps
      else
         plusd_al = one/(one-x)
      end if
   end function plus_distr_al
```

Introducing phase-space slicing parameters, these flavor kernels become: The standard flavor kernels $\overline{K}^{ab}$ are:

$$\overline{K}_\alpha^{qg}(x) = \overline{K}_\alpha^{\bar{q}g}(x) = P^{qg}(x)\log(\alpha(1-x)/x) + CF \times x \qquad (4.26)$$

$$\overline{K}_\alpha^{gq}(x) = \overline{K}_\alpha^{g\bar{q}}(x) = P^{gq}(x)\log(\alpha(1-x)/x) + TR \times 2x(1-x) \qquad (4.27)$$

$$\overline{K}_\alpha^{qq} = C_F(1-x) + P_{\text{reg}}^{qq}(x)\log\frac{\alpha(1-x)}{x} + C_F\delta(1-x)\log^2\alpha + C_F\left(\frac{2}{1-x}\log\frac{1-x}{x}\right)_+ - \left(\gamma_q \qquad (4.28)$$

⟨*SM physics: public functions*⟩+≡
```
    public :: kbarqg_al
```

⟨*SM physics: subroutines*⟩+≡
```
    function kbarqg_al (x,alpha,eps) result (kbarqgx)
      real(kind=default), intent(in) :: x, alpha, eps
      real(kind=default) :: kbarqgx
      kbarqgx = pqg (x) * log(alpha*(one-x)/x) + CF * x
    end function kbarqg_al
```

⟨*SM physics: public functions*⟩+≡
```
    public :: kbargq_al
```

⟨*SM physics: subroutines*⟩+≡
```
    function kbargq_al (x,alpha,eps) result (kbargqx)
      real(kind=default), intent(in) :: x, alpha, eps
      real(kind=default) :: kbargqx
      kbargqx = pgq (x) * log(alpha*(one-x)/x) + two * TR * x * (one-x)
    end function kbargq_al
```

⟨*SM physics: public functions*⟩+≡
```
    public :: kbarqq_al
```

⟨*SM physics: subroutines*⟩+≡
```
    function kbarqq_al (x,alpha,eps) result (kbarqqx)
       real(kind=default), intent(in) :: x, alpha, eps
       real(kind=default) :: kbarqqx
       kbarqqx = CF * (one - x) + pqq_reg(x) * log(alpha*(one-x)/x) &
                + CF * log_plus_distr(x,eps) &
                - (gamma_q + k_q_al(alpha) - CF * &
                five/6.0_default  * pi**2 - CF * (log(alpha))**2) * &
                delta(x,eps) + &
                CF * two/(one -x)*log(alpha*(two-x)/(one+alpha-x))
       if (x < (one-alpha)) then
```

```
          kbarqqx = kbarqqx - CF * two/(one-x) * log((two-x)/(one-x))
       end if
    end function kbarqq_al
```

⟨*SM physics: public functions*⟩+≡
```
    public :: kbargg_al
```

⟨*SM physics: subroutines*⟩+≡
```
    function kbargg_al (x,alpha,eps,nf) result (kbarggx)
       real(kind=default), intent(in) :: x, alpha, eps, nf
       real(kind=default) :: kbarggx
       kbarggx = pgg_reg(x) * log(alpha*(one-x)/x) &
                + CA * log_plus_distr(x,eps) &
                - (gamma_g(nf) + k_g_al(alpha,nf) - CA * &
                five/6.0_default  * pi**2 - CA * (log(alpha))**2) * &
                delta(x,eps) + &
                CA * two/(one -x)*log(alpha*(two-x)/(one+alpha-x))
       if (x < (one-alpha)) then
          kbarggx = kbarggx - CA * two/(one-x) * log((two-x)/(one-x))
       end if
    end function kbargg_al
```

The $\tilde{K}$ flavor kernels in the presence of a phase-space slicing parameter, are:

$$\tilde{K}^{ab}(x,\alpha) = P^{qq,\text{reg}}(x) \log \frac{1-x}{\alpha} + \ldots\ldots\ldots \tag{4.29}$$

⟨*SM physics: public functions*⟩+≡
```
    public :: ktildeqq_al
```

⟨*SM physics: subroutines*⟩+≡
```
    function ktildeqq_al (x,alpha,eps) result (ktildeqqx)
      real(kind=default), intent(in) :: x, eps, alpha
      real(kind=default) :: ktildeqqx
      ktildeqqx = pqq_reg(x) * log((one-x)/alpha) + CF*( &
          - log2_plus_distr_al(x,alpha,eps) - Pi**2/three * delta(x,eps) &
          + (one+x**2)/(one-x) * log(min(one,(alpha/(one-x)))) &
          + two/(one-x) * log((one+alpha-x)/alpha))
      if (x > (one-alpha)) then
         ktildeqqx = ktildeqqx - CF*two/(one-x)*log(two-x)
      end if
    end function ktildeqq_al
```

This is a logarithmic +-distribution, $\left(\frac{\log((1-x)/x)}{1-x}\right)_+$. For the sampling, we need the integral over this function over the incomplete sampling interval $[0, 1 - \epsilon]$, which is $\log^2(x) + 2Li_2(x) - \frac{\pi^2}{3}$. As this function is negative definite for $\epsilon > 0.1816$, we take a hard upper limit for that sampling parameter, irrespective of the fact what the user chooses.

⟨*SM physics: public functions*⟩+≡
```
    public :: log_plus_distr
```

⟨*SM physics: subroutines*⟩+≡
```
    function log_plus_distr (x,eps) result (lpd)
       real(kind=default), intent(in) :: x, eps
```

```
    real(kind=default) :: lpd, eps2
    eps2 = min (eps, 0.1816_default)
    if (x > (1.0_default - eps2)) then
       lpd = ((log(eps2))**2 + two*Li2(eps2) - pi**2/three)/eps2
    else
       lpd = two*log((one-x)/x)/(one-x)
    end if
end function log_plus_distr
```

Logarithmic +-distribution, $2\left(\frac{\log(1/(1-x))}{1-x}\right)_+$.

⟨*SM physics: public functions*⟩+≡
```
    public :: log2_plus_distr
```

⟨*SM physics: subroutines*⟩+≡
```
function log2_plus_distr (x,eps) result (lpd)
    real(kind=default), intent(in) :: x, eps
    real(kind=default) :: lpd
    if (x > (1.0_default - eps)) then
       lpd = - (log(eps))**2/eps
    else
       lpd = two*log(one/(one-x))/(one-x)
    end if
end function log2_plus_distr
```

Logarithmic +-distribution with phase-space slicing parameter, $2\left(\frac{\log(1/(1-x))}{1-x}\right)_{1-\alpha}$.

⟨*SM physics: public functions*⟩+≡
```
    public :: log2_plus_distr_al
```

⟨*SM physics: subroutines*⟩+≡
```
function log2_plus_distr_al (x,alpha,eps) result (lpd_al)
  real(kind=default), intent(in) :: x, eps, alpha
  real(kind=default) :: lpd_al
  if ((1.0_default - alpha) .ge. (1.0_default - eps)) then
     call msg_fatal ('alpha and epsilon chosen wrongly')
  elseif (x < (one - alpha)) then
     lpd_al = 0
  elseif (x > (1.0_default - eps)) then
     lpd_al = - ((log(eps))**2 - (log(alpha))**2)/eps
  else
     lpd_al = two*log(one/(one-x))/(one-x)
  end if
end function log2_plus_distr_al
```

Chapter 5

Physics Analysis

This part contains the structures and tools that are necessary for defining parameters, particle sets, analysis objects such as histograms, and expressions that deal with them.

These are the modules:

analysis Observables, histograms, and plots.

pdg_arrays Useful for particle aliases (e.g., 'quark' for u, d, s etc.)

prt_lists Particle lists/arrays used for analyzing events.

variables Store values of various kind, used by expressions and accessed by the command interface.

expressions Expressions of values of all kinds. Includes the API for recording analysis data.

5.1 Analysis tools

This module defines structures useful for data analysis. These include observables, histograms, and plots.

Observables are quantities that are calculated and summed up event by event. At the end, one can compute the average and error.

Histograms have their bins in addition to the observable properties. Histograms are usually written out in tables and displayed graphically.

In plots, each record creates its own entry in a table. This can be used for scatter plots if called event by event, or for plotting dependencies on parameters if called once per integration run.

⟨analysis.f90⟩≡
  ⟨*File header*⟩

  module analysis

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
    use limits, only: HISTOGRAM_HEAD_FORMAT, HISTOGRAM_DATA_FORMAT !NODEP!
    use limits, only: HISTOGRAM_INTG_FORMAT !NODEP!
  ⟨*Use file utils*⟩
    use file_utils, only: tex_format !NODEP!
    use diagnostics !NODEP!
    use os_interface

  ⟨*Standard module head*⟩

  ⟨*Analysis: public*⟩

  ⟨*Analysis: parameters*⟩

  ⟨*Analysis: types*⟩

  ⟨*Analysis: interfaces*⟩

  ⟨*Analysis: variables*⟩

  contains

  ⟨*Analysis: procedures*⟩

  end module analysis

5.1.1 Output formats

These formats share a common field width (alignment).

⟨*Limits: public parameters*⟩+≡
    character(*), parameter, public :: HISTOGRAM_HEAD_FORMAT = "1x,A13,1x"
    character(*), parameter, public :: HISTOGRAM_INTG_FORMAT = "3x,I9,3x"
    character(*), parameter, public :: HISTOGRAM_DATA_FORMAT = "1PG15.8"

5.1.2 Labels

These parameters are used for displaying data. They can be defined or reset when the driver file for the display is written.

⟨*Analysis: public*⟩≡
```
    public :: plot_labels_t
```

⟨*Analysis: types*⟩≡
```
    type :: plot_labels_t
       private
       type(string_t) :: title
       type(string_t) :: description
       type(string_t) :: xlabel
       type(string_t) :: ylabel
       integer :: width_mm = 130
       integer :: height_mm = 90
    end type plot_labels_t
```

Initialize all components that can be set by the user.

⟨*Analysis: public*⟩+≡
```
    public :: plot_labels_init
```

⟨*Analysis: procedures*⟩≡
```
    subroutine plot_labels_init (plot_labels, &
        title, description, xlabel, ylabel, width_mm, height_mm)
      type(plot_labels_t), intent(out) :: plot_labels
      type(string_t), intent(in) :: title
      type(string_t), intent(in), optional :: description
      type(string_t), intent(in), optional :: xlabel, ylabel
      integer, intent(in), optional :: width_mm, height_mm
      plot_labels%title = title
      if (present (description)) then
         plot_labels%description = description
      else
         plot_labels%description = ""
      end if
      if (present (xlabel)) then
         plot_labels%xlabel = xlabel
      else
         plot_labels%xlabel = ""
      end if
      if (present (ylabel)) then
         plot_labels%ylabel = ylabel
      else
         plot_labels%ylabel = ""
      end if
      if (present (width_mm))  plot_labels%width_mm = width_mm
      if (present (height_mm))  plot_labels%height_mm = height_mm
    end subroutine plot_labels_init
```

5.1.3 Observables

The observable type holds the accumulated observable values and weight sums which are necessary for proper averaging.

⟨Analysis: types⟩+≡

```
type :: observable_t
  private
  real(default) :: sum_values = 0
  real(default) :: sum_squared_values = 0
  real(default) :: sum_weights = 0
  real(default) :: sum_squared_weights = 0
  integer :: count = 0
  type(string_t) :: label
  type(string_t) :: physical_unit
  type(plot_labels_t) :: plot_labels
end type observable_t
```

Initialize with defined properties

⟨Analysis: procedures⟩+≡

```
subroutine observable_init (obs, label, physical_unit, plot_labels)
  type(observable_t), intent(out) :: obs
  type(string_t), intent(in), optional :: label, physical_unit
  type(plot_labels_t), intent(in), optional :: plot_labels
  if (present (label)) then
     obs%label = label
  else
     obs%label = ""
  end if
  if (present (physical_unit)) then
     obs%physical_unit = physical_unit
  else
     obs%physical_unit = ""
  end if
  if (present (plot_labels)) then
     obs%plot_labels = plot_labels
  else
     call plot_labels_init (obs%plot_labels, title = var_str ("Observable"))
  end if
end subroutine observable_init
```

Reset all numeric entries.

⟨Analysis: procedures⟩+≡

```
subroutine observable_clear (obs)
  type(observable_t), intent(inout) :: obs
  obs%sum_values = 0
  obs%sum_squared_values = 0
  obs%sum_weights = 0
  obs%sum_squared_weights = 0
  obs%count = 0
end subroutine observable_clear
```

Record a a value. Always successful for observables.

⟨Analysis: interfaces⟩≡

```
interface observable_record_value
   module procedure observable_record_value_unweighted
   module procedure observable_record_value_weighted
end interface
```

⟨*Analysis: procedures*⟩+≡

```
subroutine observable_record_value_unweighted (obs, value, success)
  type(observable_t), intent(inout) :: obs
  real(default), intent(in) :: value
  logical, intent(out), optional :: success
  obs%sum_values = obs%sum_values + value
  obs%sum_squared_values = obs%sum_squared_values + value**2
  obs%sum_weights = obs%sum_weights + 1
  obs%sum_squared_weights = obs%sum_squared_weights + 1
  obs%count = obs%count + 1
  if (present (success))  success = .true.
end subroutine observable_record_value_unweighted

subroutine observable_record_value_weighted (obs, value, weight, success)
  type(observable_t), intent(inout) :: obs
  real(default), intent(in) :: value, weight
  logical, intent(out), optional :: success
  obs%sum_values = obs%sum_values + value * weight
  obs%sum_squared_values = obs%sum_squared_values + value**2 * weight
  obs%sum_weights = obs%sum_weights + abs (weight)
  obs%sum_squared_weights = obs%sum_squared_weights + weight**2
  obs%count = obs%count + 1
  if (present (success))  success = .true.
end subroutine observable_record_value_weighted
```

Here are the statistics formulas:

1. Unweighted case: Given a sample of n values x_i, the average is

$$\langle x \rangle = \frac{\sum x_i}{n} \tag{5.1}$$

and the error estimate

$$\Delta x = \sqrt{\frac{1}{n-1} \langle \sum (x_i - \langle x \rangle)^2 } \tag{5.2}$$

$$= \sqrt{\frac{1}{n-1} \left(\frac{\sum x_i^2}{n} - \frac{(\sum x_i)^2}{n^2} \right)} \tag{5.3}$$

2. Weighted case: Instead of weight 1, each event comes with weight w_i.

$$\langle x \rangle = \frac{\sum x_i w_i}{\sum w_i} \tag{5.4}$$

and

$$\Delta x = \sqrt{\frac{1}{n-1} \left(\frac{\sum x_i^2 w_i}{\sum w_i} - \frac{(\sum x_i w_i)^2}{(\sum w_i)^2} \right)} \tag{5.5}$$

For $w_i = 1$, this specializes to the previous formula.

⟨Analysis: procedures⟩+≡

```
function observable_get_n_entries (obs) result (n)
  integer :: n
  type(observable_t), intent(in) :: obs
  n = obs%count
end function observable_get_n_entries

function observable_get_average (obs) result (avg)
  real(default) :: avg
  type(observable_t), intent(in) :: obs
  if (obs%sum_weights /= 0) then
     avg = obs%sum_values / obs%sum_weights
  else
     avg = 0
  end if
end function observable_get_average

function observable_get_error (obs) result (err)
  real(default) :: err
  type(observable_t), intent(in) :: obs
  real(default) :: var, n
  if (obs%sum_weights /= 0) then
     select case (obs%count)
     case (0:1)
        err = 0
     case default
        n = obs%count
        var = obs%sum_squared_values / obs%sum_weights &
              - (obs%sum_values / obs%sum_weights) ** 2
        err = sqrt (max (var, 0._default) / (n - 1))
     end select
  else
     err = 0
  end if
end function observable_get_error
```

Write label and/or physical unit to a string.

⟨Analysis: procedures⟩+≡

```
function observable_get_label (obs, wl, wu) result (string)
  type(string_t) :: string
  type(observable_t), intent(in) :: obs
  logical, intent(in) :: wl, wu
  type(string_t) :: label, physical_unit
  if (wl) then
     if (obs%label /= "") then
        label = obs%label
     else
        label = "\mathcal{O}"
     end if
  else
     label = ""
  end if
  if (wu) then
```

```
         if (obs%physical_unit /= "") then
            if (wl) then
               physical_unit = "\;[" // obs%physical_unit // "]"
            else
               physical_unit = obs%physical_unit
            end if
         else
            physical_unit = ""
         end if
      else
         physical_unit = ""
      end if
      string = label // physical_unit
   end function observable_get_label
```

5.1.4   Output

⟨*Analysis: procedures*⟩+≡
```
   subroutine observable_write (obs, unit)
     type(observable_t), intent(in) :: obs
     integer, intent(in), optional :: unit
     real(default) :: avg, err, relerr
     integer :: n
     integer :: u
     u = output_unit (unit);  if (u < 0)  return
     avg = observable_get_average (obs)
     err = observable_get_error (obs)
     if (avg /= 0) then
        relerr = err / avg
     else
        relerr = 0
     end if
     n = observable_get_n_entries (obs)
     write (u, "(A,1x," // HISTOGRAM_DATA_FORMAT // ")") &
          "average    =", avg
     write (u, "(A,1x," // HISTOGRAM_DATA_FORMAT // ")") &
          "error[abs]  =", err
     write (u, "(A,1x," // HISTOGRAM_DATA_FORMAT // ")") &
          "error[rel]  =", relerr
     write (u, "(A,1x," // HISTOGRAM_INTG_FORMAT // ")") &
          "n_entries   =", n
   end subroutine observable_write
```

LATEX output.

⟨*Analysis: procedures*⟩+≡
```
   subroutine observable_write_driver (obs, unit, write_heading)
     type(observable_t), intent(in) :: obs
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: write_heading
     real(default) :: avg, err
     integer :: n_digits
     logical :: heading
```

```
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      heading = .true.;  if (present (write_heading))  heading = write_heading
      avg = observable_get_average (obs)
      err = observable_get_error (obs)
      if (avg /= 0 .and. err /= 0) then
         n_digits = max (2, 2 - int (log10 (abs (err / real (avg, default)))))
      else if (avg /= 0) then
         n_digits = 100
      else
         n_digits = 1
      end if
      if (heading) then
         write (u, "(A)")
         if (obs%plot_labels%title /= "") then
            write (u, "(A)")  "\section{" // char (obs%plot_labels%title) &
                 // "}"
         else
            write (u, "(A)")  "\section{Observable}"
         end if
         if (obs%plot_labels%description /= "") then
            write (u, "(A)")  char (obs%plot_labels%description)
            write (u, *)
         end if
         write (u, "(A)")  "\begin{flushleft}"
      end if
      write (u, "(A)", advance="no")  "  $\langle{" ! $ sign
      write (u, "(A)", advance="no")  char (observable_get_label (obs, wl=.true., wu=.false.))
      write (u, "(A)", advance="no")  "}\rangle = "
      write (u, "(A)", advance="no")  char (tex_format (avg, n_digits))
      write (u, "(A)", advance="no")  "\pm"
      write (u, "(A)", advance="no")  char (tex_format (err, 2))
      write (u, "(A)", advance="no")  "\;{"
      write (u, "(A)", advance="no")  char (observable_get_label (obs, wl=.false., wu=.true.))
      write (u, "(A)")  "}"
      write (u, "(A)", advance="no")  "     \quad[n_{\text{entries}} = "
      write (u, "(I0)",advance="no")  observable_get_n_entries (obs)
      write (u, "(A)")  "]$"              ! $ fool Emacs' noweb mode
      if (heading) then
         write (u, "(A)") "\end{flushleft}"
      end if
    end subroutine observable_write_driver
```

5.1.5 Histograms

Bins

⟨*Analysis: types*⟩+≡
```
    type :: bin_t
      private
      real(default) :: midpoint = 0
      real(default) :: width = 0
      real(default) :: sum_weights = 0
```

```
      real(default) :: sum_squared_weights = 0
      real(default) :: sum_excess_weights = 0
      integer :: count = 0
   end type bin_t
```

⟨*Analysis: procedures*⟩+≡
```
   subroutine bin_init (bin, midpoint, width)
     type(bin_t), intent(out) :: bin
     real(default), intent(in) :: midpoint, width
     bin%midpoint = midpoint
     bin%width = width
   end subroutine bin_init
```

⟨*Analysis: procedures*⟩+≡
```
   elemental subroutine bin_clear (bin)
     type(bin_t), intent(inout) :: bin
     bin%sum_weights = 0
     bin%sum_squared_weights = 0
     bin%sum_excess_weights = 0
     bin%count = 0
   end subroutine bin_clear
```

⟨*Analysis: procedures*⟩+≡
```
   subroutine bin_record_value (bin, weight, excess)
     type(bin_t), intent(inout) :: bin
     real(default), intent(in) :: weight
     real(default), intent(in), optional :: excess
     bin%sum_weights = bin%sum_weights + abs (weight)
     bin%sum_squared_weights = bin%sum_squared_weights + weight ** 2
     if (present (excess)) &
          bin%sum_excess_weights = bin%sum_excess_weights + abs (excess)
     bin%count = bin%count + 1
   end subroutine bin_record_value
```

⟨*Analysis: procedures*⟩+≡
```
   function bin_get_midpoint (bin) result (x)
     real(default) :: x
     type(bin_t), intent(in) :: bin
     x = bin%midpoint
   end function bin_get_midpoint

   function bin_get_width (bin) result (w)
     real(default) :: w
     type(bin_t), intent(in) :: bin
     w = bin%width
   end function bin_get_width

   function bin_get_n_entries (bin) result (n)
     integer :: n
     type(bin_t), intent(in) :: bin
     n = bin%count
   end function bin_get_n_entries
```

```
      function bin_get_sum (bin) result (s)
        real(default) :: s
        type(bin_t), intent(in) :: bin
        s = bin%sum_weights
      end function bin_get_sum

      function bin_get_error (bin) result (err)
        real(default) :: err
        type(bin_t), intent(in) :: bin
        err = sqrt (bin%sum_squared_weights)
      end function bin_get_error

      function bin_get_excess (bin) result (excess)
        real(default) :: excess
        type(bin_t), intent(in) :: bin
        excess = bin%sum_excess_weights
      end function bin_get_excess
```

⟨*Analysis: procedures*⟩+≡

```
      subroutine bin_write_header (unit)
        integer, intent(in), optional :: unit
        character(120) :: buffer
        integer :: u
        u = output_unit (unit);  if (u < 0)  return
        write (buffer, "(A,5(1x," // HISTOGRAM_HEAD_FORMAT // "))") &
             "#", "bin midpoint", "value    ", "error    ", &
             "n_entries ", "excess      "
        write (u, "(A)")  trim (buffer)
      end subroutine bin_write_header

      subroutine bin_write (bin, unit)
        type(bin_t), intent(in) :: bin
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit);  if (u < 0)  return
        write (u, "(1x,3(1x," // HISTOGRAM_DATA_FORMAT // ")," &
                        // HISTOGRAM_INTG_FORMAT // "," &
                        // HISTOGRAM_DATA_FORMAT // ")") &
             bin_get_midpoint (bin), &
             bin_get_sum (bin), &
             bin_get_error (bin), &
             bin_get_n_entries (bin), &
             bin_get_excess (bin)
      end subroutine bin_write
```

Histograms

⟨*Analysis: types*⟩+≡

```
      type :: histogram_t
        private
        real(default) :: lower_bound = 0
        real(default) :: upper_bound = 0
```

```
    real(default) :: width = 0
    integer :: n_bins = 0
    type(observable_t) :: obs
    type(observable_t) :: obs_within_bounds
    type(bin_t) :: underflow
    type(bin_t), dimension(:), allocatable :: bin
    type(bin_t) :: overflow
    type(plot_labels_t) :: plot_labels
  end type histogram_t
```

Initializer/finalizer

Initialize a histogram. We may provide either the bin width or the number of bins.

⟨*Analysis: interfaces*⟩+≡
```
    interface histogram_init
       module procedure histogram_init_n_bins
       module procedure histogram_init_bin_width
    end interface
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine histogram_init_n_bins (h, &
         lower_bound, upper_bound, n_bins, obs_label, physical_unit, plot_labels)
      type(histogram_t), intent(out) :: h
      real(default), intent(in) :: lower_bound, upper_bound
      integer, intent(in) :: n_bins
      type(string_t), intent(in), optional :: obs_label, physical_unit
      type(plot_labels_t), intent(in), optional :: plot_labels
      real(default) :: bin_width
      integer :: i
      call observable_init (h%obs_within_bounds, obs_label, physical_unit)
      call observable_init (h%obs, obs_label, physical_unit)
      h%lower_bound = lower_bound
      h%upper_bound = upper_bound
      h%n_bins = max (n_bins, 1)
      h%width = h%upper_bound - h%lower_bound
      bin_width = h%width / h%n_bins
      allocate (h%bin (h%n_bins))
      call bin_init (h%underflow, h%lower_bound, 0._default)
      do i = 1, h%n_bins
         call bin_init (h%bin(i), &
              h%lower_bound - bin_width/2 + i * bin_width, bin_width)
      end do
      call bin_init (h%overflow, h%upper_bound, 0._default)
      if (present (plot_labels)) then
         h%plot_labels = plot_labels
      else
         call plot_labels_init (h%plot_labels, &
              title = var_str (""), xlabel = var_str (""), ylabel = var_str (""))
      end if
    end subroutine histogram_init_n_bins
```

```
subroutine histogram_init_bin_width (h, &
     lower_bound, upper_bound, bin_width, &
     obs_label, physical_unit, plot_labels)
  type(histogram_t), intent(out) :: h
  real(default), intent(in) :: lower_bound, upper_bound, bin_width
  type(string_t), intent(in), optional :: obs_label, physical_unit
  type(plot_labels_t), intent(in), optional :: plot_labels
  integer :: n_bins
  if (bin_width /= 0) then
     n_bins = nint ((upper_bound - lower_bound) / bin_width)
  else
     n_bins = 1
  end if
  call histogram_init_n_bins (h, &
       lower_bound, upper_bound, n_bins, &
       obs_label, physical_unit, plot_labels)
end subroutine histogram_init_bin_width
```

Fill histograms

Clear the histogram contents, but do not modify the structure.

⟨*Analysis: procedures*⟩+≡
```
subroutine histogram_clear (h)
  type(histogram_t), intent(inout) :: h
  call observable_clear (h%obs)
  call observable_clear (h%obs_within_bounds)
  call bin_clear (h%underflow)
  if (allocated (h%bin))  call bin_clear (h%bin)
  call bin_clear (h%overflow)
end subroutine histogram_clear
```

Record a value. Successful if the value is within bounds, otherwise it is recorded
as under-/overflow. Optionally, we may provide an excess weight that could be
returned by the unweighting procedure.

⟨*Analysis: procedures*⟩+≡
```
subroutine histogram_record_value_unweighted (h, value, excess, success)
  type(histogram_t), intent(inout) :: h
  real(default), intent(in) :: value
  real(default), intent(in), optional :: excess
  logical, intent(out), optional :: success
  integer :: i_bin
  call observable_record_value (h%obs, value)
  if (h%width /= 0) then
     i_bin = floor ((((value - h%lower_bound) / h%width) * h%n_bins) + 1
  else
     i_bin = 0
  end if
  if (i_bin <= 0) then
     call bin_record_value (h%underflow, 1._default, excess)
     if (present (success))  success = .false.
  else if (i_bin <= h%n_bins) then
     call observable_record_value (h%obs_within_bounds, value)
```

```
            call bin_record_value (h%bin(i_bin), 1._default, excess)
            if (present (success))  success = .true.
         else
            call bin_record_value (h%overflow, 1._default, excess)
            if (present (success))  success = .false.
         end if
      end subroutine histogram_record_value_unweighted
```

Weighted events: analogous, but no excess weight.

⟨*Analysis: procedures*⟩+≡
```
      subroutine histogram_record_value_weighted (h, value, weight, success)
        type(histogram_t), intent(inout) :: h
        real(default), intent(in) :: value, weight
        logical, intent(out), optional :: success
        integer :: i_bin
        call observable_record_value (h%obs, value, weight)
        if (h%width /= 0) then
           i_bin = floor ((((value - h%lower_bound) / h%width) * h%n_bins) + 1
        else
           i_bin = 0
        end if
        if (i_bin <= 0) then
           call bin_record_value (h%underflow, weight)
           if (present (success))  success = .false.
        else if (i_bin <= h%n_bins) then
           call observable_record_value (h%obs_within_bounds, weight)
           call bin_record_value (h%bin(i_bin), weight)
           if (present (success))  success = .true.
        else
           call bin_record_value (h%overflow, weight)
           if (present (success))  success = .false.
        end if
      end subroutine histogram_record_value_weighted
```

Access contents

Inherited from the observable component (all-over average etc.)

⟨*Analysis: procedures*⟩+≡
```
      function histogram_get_n_entries (h) result (n)
        integer :: n
        type(histogram_t), intent(in) :: h
        n = observable_get_n_entries (h%obs)
      end function histogram_get_n_entries

      function histogram_get_average (h) result (avg)
        real(default) :: avg
        type(histogram_t), intent(in) :: h
        avg = observable_get_average (h%obs)
      end function histogram_get_average

      function histogram_get_error (h) result (err)
        real(default) :: err
```

```
    type(histogram_t), intent(in) :: h
    err = observable_get_error (h%obs)
  end function histogram_get_error
```

Analogous, but applied only to events within bounds.

⟨Analysis: procedures⟩+≡
```
  function histogram_get_n_entries_within_bounds (h) result (n)
    integer :: n
    type(histogram_t), intent(in) :: h
    n = observable_get_n_entries (h%obs_within_bounds)
  end function histogram_get_n_entries_within_bounds

  function histogram_get_average_within_bounds (h) result (avg)
    real(default) :: avg
    type(histogram_t), intent(in) :: h
    avg = observable_get_average (h%obs_within_bounds)
  end function histogram_get_average_within_bounds

  function histogram_get_error_within_bounds (h) result (err)
    real(default) :: err
    type(histogram_t), intent(in) :: h
    err = observable_get_error (h%obs_within_bounds)
  end function histogram_get_error_within_bounds
```

Check bins. If the index is zero or above the limit, return the results for underflow or overflow, respectively.

⟨Analysis: procedures⟩+≡
```
  function histogram_get_n_entries_for_bin (h, i) result (n)
    integer :: n
    type(histogram_t), intent(in) :: h
    integer, intent(in) :: i
    if (i <= 0) then
       n = bin_get_n_entries (h%underflow)
    else if (i <= h%n_bins) then
       n = bin_get_n_entries (h%bin(i))
    else
       n = bin_get_n_entries (h%overflow)
    end if
  end function histogram_get_n_entries_for_bin

  function histogram_get_sum_for_bin (h, i) result (avg)
    real(default) :: avg
    type(histogram_t), intent(in) :: h
    integer, intent(in) :: i
    if (i <= 0) then
       avg = bin_get_sum (h%underflow)
    else if (i <= h%n_bins) then
       avg = bin_get_sum (h%bin(i))
    else
       avg = bin_get_sum (h%overflow)
    end if
  end function histogram_get_sum_for_bin
```

```
function histogram_get_error_for_bin (h, i) result (err)
  real(default) :: err
  type(histogram_t), intent(in) :: h
  integer, intent(in) :: i
  if (i <= 0) then
     err = bin_get_error (h%underflow)
  else if (i <= h%n_bins) then
     err = bin_get_error (h%bin(i))
  else
     err = bin_get_error (h%overflow)
  end if
end function histogram_get_error_for_bin

function histogram_get_excess_for_bin (h, i) result (err)
  real(default) :: err
  type(histogram_t), intent(in) :: h
  integer, intent(in) :: i
  if (i <= 0) then
     err = bin_get_excess (h%underflow)
  else if (i <= h%n_bins) then
     err = bin_get_excess (h%bin(i))
  else
     err = bin_get_excess (h%overflow)
  end if
end function histogram_get_excess_for_bin
```

Output

⟨*Analysis: procedures*⟩+≡

```
subroutine histogram_write (h, unit)
  type(histogram_t), intent(in) :: h
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit);  if (u < 0)  return
  call bin_write_header (u)
  if (allocated (h%bin)) then
     do i = 1, h%n_bins
        call bin_write (h%bin(i), u)
     end do
  end if
  write (u, *)
  write (u, "(A,1x,A)")  "#", "Underflow:"
  call bin_write (h%underflow, u)
  write (u, *)
  write (u, "(A,1x,A)")  "#", "Overflow:"
  call bin_write (h%overflow, u)
  write (u, *)
  write (u, "(A,1x,A)")  "#", "Summary: data within bounds"
  call observable_write (h%obs_within_bounds, u)
  write (u, *)
  write (u, "(A,1x,A)")  "#", "Summary: all data"
  call observable_write (h%obs, u)
  write (u, *)
```

```
      end subroutine histogram_write
```

LaTeX output.

⟨*Analysis: procedures*⟩+≡
```
      subroutine histogram_write_driver (h, filename, unit, write_heading)
        type(histogram_t), intent(in) :: h
        type(string_t), intent(in) :: filename
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: write_heading
        logical :: heading
        character(len=32) :: lower_bound_str, upper_bound_str, bin_half_width_str
        logical, parameter :: log_plot = .false.
        integer :: u
        u = output_unit (unit);  if (u < 0)  return
        heading = .true.;  if (present (write_heading))  heading = write_heading
        if (heading) then
           write (u, "(A)")
           if (h%plot_labels%title /= "") then
              write (u, "(A)")  "\section{" // char (h%plot_labels%title) // "}"
           else
              write (u, "(A)")  "\section{Histogram}"
           end if
        end if
        if (h%plot_labels%description /= "") then
           write (u, "(A)")  char (h%plot_labels%description)
           write (u, *)
           write (u, "(A)")  "\vspace*{\baselineskip}"
        end if
        lower_bound_str = ""
        upper_bound_str = ""
        bin_half_width_str = ""
1       format (G10.3)
        write (lower_bound_str, 1)  h%lower_bound
        write (upper_bound_str, 1)  h%upper_bound
        write (bin_half_width_str,  1)  h%width / h%n_bins / 2
        lower_bound_str = adjustl (lower_bound_str)
        upper_bound_str = adjustl (upper_bound_str)
        bin_half_width_str = adjustl (bin_half_width_str)
        write (u, "(A)")  "\vspace*{\baselineskip}"
        write (u, "(A)")  "\unitlength 1mm"
        write (u, "(A,I0,',',I0,A)")  &
             "\begin{gmlgraph*}(", &
             h%plot_labels%width_mm, h%plot_labels%height_mm, &
             ")[dat]"
        if (log_plot) then
           write (u, "(2x,A)")  "setup (linear,log); "
           write (u, "(2x,A)")  &
                "graphrange (#" // trim (lower_bound_str) // ", ??), " &
                //          "(#" // trim (upper_bound_str) // ", ??);"
        else
           write (u, "(2x,A)")  "setup (linear,linear); "
           write (u, "(2x,A)")  &
                "graphrange (#" // trim (lower_bound_str) // ", #0), " &
```

```
                    //            "(#" // trim (upper_bound_str) // ", ??);"
        end if
        write (u, "(2x,A)")  'fromfile "' // char (filename) // '":'
        write (u, "(4x,A)")  'key "# Histogram:";'
        write (u, "(4x,A)")  'dx := #' // trim (bin_half_width_str) // ';'
        write (u, "(4x,A)")  'for i withinblock:'
        write (u, "(6x,A)")  'get x, y, y.d, n, y.e;'
        write (u, "(6x,A)")  'plot (dat) (x,y) hbar dx;'
!       write (u, "(6x,A)")  'if show_excess: ' // &
!                    & 'plot(dat.e)(x, y plus y.e) hbar dx; fi'
        write (u, "(4x,A)")  'endfor'
        write (u, "(4x,A)")  'calculate dat.base (dat) (x,#0);'
        write (u, "(2x,A)")  'endfrom'
        write (u, "(2x,A)")  'fill piecewise from (dat, dat.base/\) ' &
            // 'withcolor col.default outlined;'
!       write (u, "(2x,A)") 'if show_excess: ' // &
!                    & 'fill piecewise from(dat.e, dat/\) '// &
!                    & 'withcolor col.excess outlined; fi'
!       if (mcs%normalize_weight) then
        if (h%plot_labels%ylabel /= "") then
           write (u, "(2x,A)")  'label.ulft (<' // '<' &
               // char (h%plot_labels%ylabel) // '>' // '>, out);'
        else
           write (u, "(2x,A)")  'label.ulft (<' // '<\#evt/bin>' // '>, out);'
        end if
!       else
!          write(u, '(2x,A)') 'label.ulft(<'//'<$d\sigma\,[{\rm fb}]$/bin>'//'>, out);'
!       end if
        if (h%plot_labels%xlabel /= "") then
           write (u, "(2x,A)", advance="no")  'label.bot (<' // '<' &
               // char (h%plot_labels%xlabel) // '>' &
               // '>, out);'
        else
           write (u, "(2x,A)", advance="no")  'label.bot (<' // '<${'
           write (u, "(A)", advance="no")  &
               char (observable_get_label (h%obs, wl=.true., wu=.true.))
           write (u, "(A)")  '}$>' // '>, out);'
        end if
        write (u, "(A)") "\end{gmlgraph*}"
        write (u, "(A)") "\vspace*{2\baselineskip}"
        write (u, "(A)") "\begin{flushleft}"
        write (u, "(A)") "\textbf{Data within bounds:} \\"
        call observable_write_driver (h%obs_within_bounds, unit, &
                                      write_heading=.false.)
        write (u, "(A)") "\\[0.5\baselineskip]"
        write (u, "(A)") "\textbf{All data:} \\"
        call observable_write_driver (h%obs, unit, write_heading=.false.)
        write (u, "(A)") "\end{flushleft}"
    end subroutine histogram_write_driver
```

241

5.1.6 Plots

Points

⟨*Analysis: types*⟩+≡
```
type :: point_t
  private
  real(default) :: x = 0
  real(default) :: y = 0
  real(default) :: xerr = 0
  real(default) :: yerr = 0
  type(point_t), pointer :: next => null ()
end type point_t
```

⟨*Analysis: procedures*⟩+≡
```
subroutine point_init (point, x, y, xerr, yerr)
  type(point_t), intent(out) :: point
  real(default), intent(in) :: x, y
  real(default), intent(in), optional :: xerr, yerr
  point%x = x
  point%y = y
  if (present (xerr))  point%xerr = xerr
  if (present (yerr))  point%yerr = yerr
end subroutine point_init
```

⟨*Analysis: procedures*⟩+≡
```
function point_get_x (point) result (x)
  real(default) :: x
  type(point_t), intent(in) :: point
  x = point%x
end function point_get_x

function point_get_y (point) result (y)
  real(default) :: y
  type(point_t), intent(in) :: point
  y = point%y
end function point_get_y

function point_get_xerr (point) result (xerr)
  real(default) :: xerr
  type(point_t), intent(in) :: point
  xerr = point%xerr
end function point_get_xerr

function point_get_yerr (point) result (yerr)
  real(default) :: yerr
  type(point_t), intent(in) :: point
  yerr = point%yerr
end function point_get_yerr
```

⟨*Analysis: procedures*⟩+≡
```
subroutine point_write_header (unit)
  integer, intent(in) :: unit
  character(120) :: buffer
```

```
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (buffer, "(A,4(1x," // HISTOGRAM_HEAD_FORMAT // "))") &
            "#", "x        ", "y        ", "xerr     ", "yerr     "
      write (u, "(A)")  trim (buffer)
    end subroutine point_write_header

    subroutine point_write (point, unit)
      type(point_t), intent(in) :: point
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,4(1x," // HISTOGRAM_DATA_FORMAT // ")),") &
            point_get_x (point), &
            point_get_y (point), &
            point_get_xerr (point), &
            point_get_yerr (point)
    end subroutine point_write
```

Histograms

⟨*Analysis: types*⟩+≡
```
    type :: plot_t
       private
       real(default) :: lower_bound = 0
       real(default) :: upper_bound = 0
       real(default) :: width = 0
       type(bin_t) :: underflow
       type(point_t), pointer :: first => null ()
       type(point_t), pointer :: last => null ()
       type(bin_t) :: overflow
       integer :: count = 0
       integer :: count_within_bounds = 0
       type(plot_labels_t) :: plot_labels
    end type plot_t
```

Initializer/finalizer

Initialize a plot. We provide the lower and upper bound in the x direction.

⟨*Analysis: procedures*⟩+≡
```
    subroutine plot_init (plot, lower_bound, upper_bound, plot_labels)
      type(plot_t), intent(out) :: plot
      real(default), intent(in) :: lower_bound, upper_bound
      type(plot_labels_t), intent(in), optional :: plot_labels
      plot%lower_bound = lower_bound
      plot%upper_bound = upper_bound
      plot%width = plot%upper_bound - plot%lower_bound
      call bin_init (plot%underflow, plot%lower_bound, 0._default)
      call bin_init (plot%overflow, plot%upper_bound, 0._default)
      if (present (plot_labels)) then
         plot%plot_labels = plot_labels
```

```
          else
             call plot_labels_init (plot%plot_labels, &
                   title = var_str (""), xlabel = var_str (""), ylabel = var_str (""))
          end if
       end subroutine plot_init
```

Finalize the plot by deallocating the list of points.

⟨*Analysis: procedures*⟩+≡
```
       subroutine plot_final (plot)
         type(plot_t), intent(inout) :: plot
         type(point_t), pointer :: current
         do while (associated (plot%first))
            current => plot%first
            plot%first => current%next
            deallocate (current)
         end do
         plot%last => null ()
       end subroutine plot_final
```

Fill plots

Clear the plot contents, but do not modify the structure.

⟨*Analysis: procedures*⟩+≡
```
       subroutine plot_clear (plot)
         type(plot_t), intent(inout) :: plot
         call bin_clear (plot%underflow)
         call bin_clear (plot%overflow)
         plot%count = 0
         plot%count_within_bounds = 0
         call plot_final (plot)
       end subroutine plot_clear
```

Record a value. Successful if the value is within bounds, otherwise it is recorded as under-/overflow.

⟨*Analysis: procedures*⟩+≡
```
       subroutine plot_record_value (plot, x, y, xerr, yerr, success)
         type(plot_t), intent(inout) :: plot
         real(default), intent(in) :: x, y
         real(default), intent(in), optional :: xerr, yerr
         logical, intent(out), optional :: success
         type(point_t), pointer :: point
         plot%count = plot%count + 1
         if (x < plot%lower_bound) then
            call bin_record_value (plot%underflow, 1._default)
            if (present (success))  success = .false.
         else if (x <= plot%upper_bound) then
            plot%count_within_bounds = plot%count_within_bounds + 1
            allocate (point)
            call point_init (point, x, y, xerr, yerr)
            if (associated (plot%first)) then
               plot%last%next => point
```

```
      else
         plot%first => point
      end if
      plot%last => point
      if (present (success))  success = .true.
   else
      call bin_record_value (plot%overflow, 1._default)
      if (present (success))  success = .false.
   end if
end subroutine plot_record_value
```

Access contents

The number of entries total, and the number of entries that have been recorded:

⟨Analysis: procedures⟩+≡
```
   function plot_get_n_entries (plot) result (n)
     integer :: n
     type(plot_t), intent(in) :: plot
     n = plot%count
   end function plot_get_n_entries
```

Analogous, but applied only to events within bounds.

⟨Analysis: procedures⟩+≡
```
   function plot_get_n_entries_within_bounds (plot) result (n)
     integer :: n
     type(plot_t), intent(in) :: plot
     n = plot%count_within_bounds
   end function plot_get_n_entries_within_bounds
```

Output

⟨Analysis: procedures⟩+≡
```
   subroutine plot_write (plot, unit)
     type(plot_t), intent(in) :: plot
     integer, intent(in), optional :: unit
     type(point_t), pointer :: point
     integer :: u
     u = output_unit (unit);  if (u < 0)  return
     call point_write_header (u)
     point => plot%first
     do while (associated (point))
        call point_write (point, unit)
        point => point%next
     end do
     write (u, *)
     call bin_write_header (u)
     write (u, "(A,1x,A)")  "#", "Underflow:"
     call bin_write (plot%underflow, u)
     write (u, *)
     write (u, "(A,1x,A)")  "#", "Overflow:"
     call bin_write (plot%overflow, u)
```

```
      write (u, *)
      write (u, "(A,1x,A)")  "#", "Summary: points within bounds"
      write (u, "(A," // HISTOGRAM_INTG_FORMAT // ")") &
            "n_entries = ", plot_get_n_entries_within_bounds (plot)
      write (u, *)
      write (u, "(A,1x,A)")  "#", "Summary: all points"
      write (u, "(A," // HISTOGRAM_INTG_FORMAT // ")") &
            "n_entries = ", plot_get_n_entries (plot)
      write (u, *)
    end subroutine plot_write
```

LaTeX output.

⟨*Analysis: procedures*⟩+≡

```
    subroutine plot_write_driver (plot, filename, unit, write_heading)
      type(plot_t), intent(in) :: plot
      type(string_t), intent(in) :: filename
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: write_heading
      logical :: heading
      character(len=32) :: lower_bound_str, upper_bound_str
      logical, parameter :: log_plot = .false.
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      heading = .true.;  if (present (write_heading))  heading = write_heading
      if (heading) then
         write (u, "(A)")
         if (plot%plot_labels%title /= "") then
            write (u, "(A)")  "\section{" // char (plot%plot_labels%title) // "}"
         else
            write (u, "(A)")  "\section{Plot}"
         end if
      end if
      if (plot%plot_labels%description /= "") then
         write (u, "(A)")  char (plot%plot_labels%description)
         write (u, *)
         write (u, "(A)")  "\vspace*{\baselineskip}"
      end if
      lower_bound_str = ""
      upper_bound_str = ""
  1   format (G10.3)
      write (lower_bound_str, 1)  plot%lower_bound
      write (upper_bound_str, 1)  plot%upper_bound
      lower_bound_str = adjustl (lower_bound_str)
      upper_bound_str = adjustl (upper_bound_str)
      write (u, "(A)")  "\vspace*{\baselineskip}"
      write (u, "(A)")  "\unitlength 1mm"
      write (u, "(A,I0,',',I0,A)")  &
            "\begin{gmlgraph*}(", &
            plot%plot_labels%width_mm, plot%plot_labels%height_mm, &
            ")[dat]"
      if (log_plot) then
         write (u, "(2x,A)")  "setup (linear,log); "
         write (u, "(2x,A)")  &
```

246

```
          "graphrange (#" // trim (lower_bound_str) // ", ??), " &
          //          "(#" // trim (upper_bound_str) // ", ??);"
   else
      write (u, "(2x,A)")  "setup (linear,linear); "
      write (u, "(2x,A)")  &
          "graphrange (#" // trim (lower_bound_str) // ", #0), " &
          //          "(#" // trim (upper_bound_str) // ", ??);"
   end if
   write (u, "(2x,A)")  'fromfile "' // char (filename) // '":'
   write (u, "(4x,A)")  'key "# Plot:";'
   write (u, "(4x,A)")  'for i withinblock:'
   write (u, "(6x,A)")  'get x, y, x.err, y.err;'
   write (u, "(6x,A)")  'plot (dat) (x,y);'
   write (u, "(4x,A)")  'endfor'
   write (u, "(2x,A)")  'endfrom'
   write (u, "(2x,A)")  'draw from (dat); '
   if (plot%plot_labels%ylabel /= "") then
      write (u, "(2x,A)")  'label.ulft (<' // '<' &
          // char (plot%plot_labels%ylabel) // '>' // '>, out);'
   else
      write (u, "(2x,A)")  'label.ulft (<' // '<y>' // '>, out);'
   end if
   if (plot%plot_labels%xlabel /= "") then
      write (u, "(2x,A)", advance="no")  'label.bot (<' // '<' &
          // char (plot%plot_labels%xlabel) // '>' &
          // '>, out);'
   else
      write (u, "(2x,A)")  'label.ulft (<' // '<x>' // '>, out);'
   end if
   write (u, "(A)")  "\end{gmlgraph*}"
   write (u, "(A)")  "\vspace*{2\baselineskip}"
   write (u, "(A)")  "\begin{flushleft}"
   write (u, "(A)")  "\textbf{Data within bounds:} \\"
   write (u, "(A)", advance="no")  "$n_{\text{entries}} = "
   write (u, "(I0,A)",advance="no")  plot%count_within_bounds, "$"
   write (u, "(A)")  "\\[0.5\baselineskip]"
   write (u, "(A)")  "\textbf{All data:} \\"
   write (u, "(A)", advance="no")  "$n_{\text{entries}} = "
   write (u, "(I0,A)")  plot%count, "$"
   write (u, "(A)")  "\end{flushleft}"
end subroutine plot_write_driver
```

5.1.7 Analysis objects

This data structure holds all observables, histograms and such that are currently
active. We have one global store; individual items are identified by their ID
strings.

(This should rather be coded by type extension.)

⟨*Analysis: parameters*⟩≡

```
integer, parameter :: AN_UNDEFINED = 0
integer, parameter :: AN_OBSERVABLE = 1
integer, parameter :: AN_HISTOGRAM = 2
```

```
      integer, parameter :: AN_PLOT = 3
```

⟨*Analysis: types*⟩+≡
```
    type :: analysis_object_t
      private
      type(string_t) :: id
      integer :: type = AN_UNDEFINED
      type(observable_t), pointer :: obs => null ()
      type(histogram_t), pointer :: h => null ()
      type(plot_t), pointer :: plot => null ()
      type(analysis_object_t), pointer :: next => null ()
    end type analysis_object_t
```

Initializer/finalizer

Allocate with the correct type but do not fill initial values.

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_object_init (obj, id, type)
      type(analysis_object_t), intent(out) :: obj
      type(string_t), intent(in) :: id
      integer, intent(in) :: type
      obj%id = id
      obj%type = type
      select case (obj%type)
      case (AN_OBSERVABLE);  allocate (obj%obs)
      case (AN_HISTOGRAM);   allocate (obj%h)
      case (AN_PLOT);        allocate (obj%plot)
      end select
    end subroutine analysis_object_init
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_object_final (obj)
      type(analysis_object_t), intent(inout) :: obj
      select case (obj%type)
      case (AN_OBSERVABLE)
         deallocate (obj%obs)
      case (AN_HISTOGRAM)
         deallocate (obj%h)
      case (AN_PLOT)
         call plot_final (obj%plot)
         deallocate (obj%plot)
      end select
      obj%type = AN_UNDEFINED
    end subroutine analysis_object_final
```

Clear the analysis object, i.e., reset it to its initial state.

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_object_clear (obj)
      type(analysis_object_t), intent(inout) :: obj
      select case (obj%type)
      case (AN_OBSERVABLE)
```

```
         call observable_clear (obj%obs)
      case (AN_HISTOGRAM)
         call histogram_clear (obj%h)
      case (AN_PLOT)
         call plot_clear (obj%plot)
      end select
    end subroutine analysis_object_clear
```

Fill with data

Record data. The effect depends on the type of analysis object.

⟨*Analysis: procedures*⟩+≡

```
    subroutine analysis_object_record_data (obj, &
        x, y, xerr, yerr, weight, excess, success)
      type(analysis_object_t), intent(inout) :: obj
      real(default), intent(in) :: x
      real(default), intent(in), optional :: y, xerr, yerr, weight, excess
      logical, intent(out), optional :: success
      select case (obj%type)
      case (AN_OBSERVABLE)
         if (present (weight)) then
            call observable_record_value_weighted (obj%obs, x, weight, success)
         else
            call observable_record_value_unweighted (obj%obs, x, success)
         end if
      case (AN_HISTOGRAM)
         if (present (weight)) then
            call histogram_record_value_weighted (obj%h, x, weight, success)
         else
            call histogram_record_value_unweighted (obj%h, x, excess, success)
         end if
      case (AN_PLOT)
         if (present (y)) then
            call plot_record_value (obj%plot, x, y, xerr, yerr, success)
         else
            if (present (success))  success = .false.
         end if
      case default
         if (present (success))  success = .false.
      end select
    end subroutine analysis_object_record_data
```

Explicitly set the pointer to the next object in the list.

⟨*Analysis: procedures*⟩+≡

```
    subroutine analysis_object_set_next_ptr (obj, next)
      type(analysis_object_t), intent(inout) :: obj
      type(analysis_object_t), pointer :: next
      obj%next => next
    end subroutine analysis_object_set_next_ptr
```

Access contents

Return a pointer to the next object in the list.

⟨*Analysis: procedures*⟩+≡
```
function analysis_object_get_next_ptr (obj) result (next)
  type(analysis_object_t), pointer :: next
  type(analysis_object_t), intent(in) :: obj
  next => obj%next
end function analysis_object_get_next_ptr
```

Return generic entries.

⟨*Analysis: procedures*⟩+≡
```
function analysis_object_get_n_entries (obj, within_bounds) result (n)
  integer :: n
  type(analysis_object_t), intent(in) :: obj
  logical, intent(in), optional :: within_bounds
  logical :: wb
  select case (obj%type)
  case (AN_OBSERVABLE)
     n = observable_get_n_entries (obj%obs)
  case (AN_HISTOGRAM)
     wb = .false.;  if (present (within_bounds)) wb = within_bounds
     if (wb) then
        n = histogram_get_n_entries_within_bounds (obj%h)
     else
        n = histogram_get_n_entries (obj%h)
     end if
  case (AN_PLOT)
     wb = .false.;  if (present (within_bounds)) wb = within_bounds
     if (wb) then
        n = plot_get_n_entries_within_bounds (obj%plot)
     else
        n = plot_get_n_entries (obj%plot)
     end if
  case default
     n = 0
  end select
end function analysis_object_get_n_entries

function analysis_object_get_average (obj, within_bounds) result (avg)
  real(default) :: avg
  type(analysis_object_t), intent(in) :: obj
  logical, intent(in), optional :: within_bounds
  logical :: wb
  select case (obj%type)
  case (AN_OBSERVABLE)
     avg = observable_get_average (obj%obs)
  case (AN_HISTOGRAM)
     wb = .false.;  if (present (within_bounds)) wb = within_bounds
     if (wb) then
        avg = histogram_get_average_within_bounds (obj%h)
     else
        avg = histogram_get_average (obj%h)
     end if
```

```
      case default
         avg = 0
      end select
    end function analysis_object_get_average

    function analysis_object_get_error (obj, within_bounds) result (err)
      real(default) :: err
      type(analysis_object_t), intent(in) :: obj
      logical, intent(in), optional :: within_bounds
      logical :: wb
      select case (obj%type)
      case (AN_OBSERVABLE)
         err = observable_get_error (obj%obs)
      case (AN_HISTOGRAM)
         wb = .false.;  if (present (within_bounds)) wb = within_bounds
         if (wb) then
            err = histogram_get_error_within_bounds (obj%h)
         else
            err = histogram_get_error (obj%h)
         end if
      case default
         err = 0
      end select
    end function analysis_object_get_error
```

Return pointers to the actual contents:

⟨*Analysis: procedures*⟩+≡

```
    function analysis_object_get_observable_ptr (obj) result (obs)
      type(observable_t), pointer :: obs
      type(analysis_object_t), intent(in) :: obj
      select case (obj%type)
      case (AN_OBSERVABLE);  obs => obj%obs
      case default;          obs => null ()
      end select
    end function analysis_object_get_observable_ptr

    function analysis_object_get_histogram_ptr (obj) result (h)
      type(histogram_t), pointer :: h
      type(analysis_object_t), intent(in) :: obj
      select case (obj%type)
      case (AN_HISTOGRAM);  h => obj%h
      case default;         h => null ()
      end select
    end function analysis_object_get_histogram_ptr

    function analysis_object_get_plot_ptr (obj) result (plot)
      type(plot_t), pointer :: plot
      type(analysis_object_t), intent(in) :: obj
      select case (obj%type)
      case (AN_PLOT);  plot => obj%plot
      case default;    plot => null ()
      end select
    end function analysis_object_get_plot_ptr
```

Return true if the object has a graphical representation:

⟨*Analysis: procedures*⟩+≡

```
function analysis_object_has_plot (obj) result (flag)
  logical :: flag
  type(analysis_object_t), intent(in) :: obj
  select case (obj%type)
  case (AN_HISTOGRAM);  flag = .true.
  case (AN_PLOT);       flag = .true.
  case default;         flag = .false.
  end select
end function analysis_object_has_plot
```

Output

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_object_write (obj, unit)
  type(analysis_object_t), intent(in) :: obj
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(A)")  repeat ("#", 79)
  select case (obj%type)
  case (AN_OBSERVABLE)
    write (u, "(A)", advance="no")  "# Observable:"
  case (AN_HISTOGRAM)
    write (u, "(A)", advance="no")  "# Histogram: "
  case (AN_PLOT)
    write (u, "(A)", advance="no")  "# Plot: "
  case default
    write (u, "(A)") "# [undefined analysis object]"
    return
  end select
  write (u, "(1x,A)")  char (obj%id)
  select case (obj%type)
  case (AN_OBSERVABLE);  call observable_write (obj%obs, unit)
  case (AN_HISTOGRAM);   call histogram_write (obj%h, unit)
  case (AN_PLOT);        call plot_write (obj%plot, unit)
  end select
end subroutine analysis_object_write
```

Write the object part of the LaTeX driver file.

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_object_write_driver (obj, filename, unit)
  type(analysis_object_t), intent(in) :: obj
  type(string_t), intent(in) :: filename
  integer, intent(in), optional :: unit
  select case (obj%type)
  case (AN_OBSERVABLE);  call observable_write_driver (obj%obs, unit)
  case (AN_HISTOGRAM);   call histogram_write_driver (obj%h, filename, unit)
  case (AN_PLOT);        call plot_write_driver (obj%plot, filename, unit)
  end select
end subroutine analysis_object_write_driver
```

5.1.8 Analysis store

This data structure holds all observables, histograms and such that are currently active. We have one global store; individual items are identified by their ID strings and types.

⟨*Analysis: variables*⟩≡
```
type(analysis_store_t), save :: analysis_store
```

⟨*Analysis: types*⟩+≡
```
type :: analysis_store_t
  private
  type(analysis_object_t), pointer :: first => null ()
  type(analysis_object_t), pointer :: last => null ()
end type analysis_store_t
```

Delete the analysis store

⟨*Analysis: public*⟩+≡
```
public :: analysis_final
```

⟨*Analysis: procedures*⟩+≡
```
subroutine analysis_final ()
  type(analysis_object_t), pointer :: current
  do while (associated (analysis_store%first))
     current => analysis_store%first
     analysis_store%first => current%next
     call analysis_object_final (current)
  end do
  analysis_store%last => null ()
end subroutine analysis_final
```

Append a new analysis object

⟨*Analysis: procedures*⟩+≡
```
subroutine analysis_store_append_object (id, type)
  type(string_t), intent(in) :: id
  integer, intent(in) :: type
  type(analysis_object_t), pointer :: obj
  allocate (obj)
  call analysis_object_init (obj, id, type)
  if (associated (analysis_store%last)) then
     analysis_store%last%next => obj
  else
     analysis_store%first => obj
  end if
  analysis_store%last => obj
end subroutine analysis_store_append_object
```

Return a pointer to the analysis object with given ID.

⟨*Analysis: procedures*⟩+≡
```
function analysis_store_get_object_ptr (id) result (obj)
```

```
      type(string_t), intent(in) :: id
      type(analysis_object_t), pointer :: obj
      obj => analysis_store%first
      do while (associated (obj))
         if (obj%id == id)  return
         obj => obj%next
      end do
   end function analysis_store_get_object_ptr
```

Initialize an analysis object: either reset it if present, or append a new entry.

⟨*Analysis: procedures*⟩+≡
```
   subroutine analysis_store_init_object (id, type, obj)
     type(string_t), intent(in) :: id
     integer, intent(in) :: type
     type(analysis_object_t), pointer :: obj, next
     obj => analysis_store_get_object_ptr (id)
     if (associated (obj)) then
        next => analysis_object_get_next_ptr (obj)
        call analysis_object_final (obj)
        call analysis_object_init (obj, id, type)
        call analysis_object_set_next_ptr (obj, next)
     else
        call analysis_store_append_object (id, type)
        obj => analysis_store%last
     end if
   end subroutine analysis_store_init_object
```

5.1.9   LaTeX driver file

Write a driver file for all objects in the store.

⟨*Analysis: procedures*⟩+≡
```
   subroutine analysis_store_write_driver_all (filename_data, unit)
     type(string_t), intent(in) :: filename_data
     integer, intent(in), optional :: unit
     type(analysis_object_t), pointer :: obj
     call analysis_store_write_driver_header (unit)
     obj => analysis_store%first
     do while (associated (obj))
        call analysis_object_write_driver (obj, filename_data, unit)
        obj => obj%next
     end do
     call analysis_store_write_driver_footer (unit)
   end subroutine analysis_store_write_driver_all
```

Write a driver file for an array of objects.

⟨*Analysis: procedures*⟩+≡
```
   subroutine analysis_store_write_driver_obj (filename_data, id, unit)
     type(string_t), intent(in) :: filename_data
     type(string_t), dimension(:), intent(in) :: id
     integer, intent(in), optional :: unit
     type(analysis_object_t), pointer :: obj
```

```
      integer :: i
      call analysis_store_write_driver_header (unit)
      do i = 1, size (id)
         obj => analysis_store_get_object_ptr (id(i))
         if (associated (obj))  &
              call analysis_object_write_driver (obj, filename_data, unit)
      end do
      call analysis_store_write_driver_footer (unit)
   end subroutine analysis_store_write_driver_obj
```

The beginning of the driver file.

⟨*Analysis: procedures*⟩+≡

```
   subroutine analysis_store_write_driver_header (unit)
     integer, intent(in), optional :: unit
     integer :: u
     u = output_unit (unit);  if (u < 0)  return
     write(u, '(A)') "\documentclass[12pt]{article}"
     write(u, *)
     write(u, '(A)') "\usepackage{gamelan}"
     write(u, '(A)') "\usepackage{amsmath}"
     write(u, *)
     write(u, '(A)') "\begin{document}"
     write(u, '(A)') "\begin{gmlfile}"
     write(u, *)
     write(u, '(A)') "\begin{gmlcode}"
     write(u, '(A)') "  color col.default, col.excess;"
     write(u, '(A)') "  col.default = 0.9white;"
     write(u, '(A)') "  col.excess  = red;"
     write(u, '(A)') "  boolean show_excess;"
 !    if (mcs(1)%plot_excess .and. mcs(1)%unweighted) then
 !       write(u, '(A)') "  show_excess = true;"
 !    else
     write(u, '(A)') "  show_excess = false;"
 !    end if
     write(u, '(A)') "\end{gmlcode}"
     write(u, *)
   end subroutine analysis_store_write_driver_header
```

The end of the driver file.

⟨*Analysis: procedures*⟩+≡

```
   subroutine analysis_store_write_driver_footer (unit)
     integer, intent(in), optional :: unit
     integer :: u
     u = output_unit (unit);  if (u < 0)  return
     write(u, *)
     write(u, '(A)') "\end{gmlfile}"
     write(u, '(A)') "\end{document}"
   end subroutine analysis_store_write_driver_footer
```

5.1.10 API

Creating new objects

The specific versions below:

⟨*Analysis: public*⟩+≡

```
    public :: analysis_init_observable
```

⟨*Analysis: procedures*⟩+≡

```
    subroutine analysis_init_observable (id, label, physical_unit, plot_labels)
      type(string_t), intent(in) :: id
      type(string_t), intent(in), optional :: label, physical_unit
      type(plot_labels_t), intent(in), optional :: plot_labels
      type(analysis_object_t), pointer :: obj
      type(observable_t), pointer :: obs
      call analysis_store_init_object (id, AN_OBSERVABLE, obj)
      obs => analysis_object_get_observable_ptr (obj)
      call observable_init (obs, label, physical_unit, plot_labels)
    end subroutine analysis_init_observable
```

⟨*Analysis: public*⟩+≡

```
    public :: analysis_init_histogram
```

⟨*Analysis: interfaces*⟩+≡

```
    interface analysis_init_histogram
       module procedure analysis_init_histogram_n_bins
       module procedure analysis_init_histogram_bin_width
    end interface
```

⟨*Analysis: procedures*⟩+≡

```
    subroutine analysis_init_histogram_n_bins &
        (id, lower_bound, upper_bound, n_bins, &
         label, physical_unit, plot_labels)
      type(string_t), intent(in) :: id
      real(default), intent(in) :: lower_bound, upper_bound
      integer, intent(in) :: n_bins
      type(string_t), intent(in), optional :: label, physical_unit
      type(plot_labels_t), intent(in), optional :: plot_labels
      type(analysis_object_t), pointer :: obj
      type(histogram_t), pointer :: h
      call analysis_store_init_object (id, AN_HISTOGRAM, obj)
      h => analysis_object_get_histogram_ptr (obj)
      call histogram_init (h, &
          lower_bound, upper_bound, n_bins, &
          label, physical_unit, plot_labels)
    end subroutine analysis_init_histogram_n_bins

    subroutine analysis_init_histogram_bin_width &
        (id, lower_bound, upper_bound, bin_width, &
         label, physical_unit, plot_labels)
      type(string_t), intent(in) :: id
      real(default), intent(in) :: lower_bound, upper_bound, bin_width
      type(string_t), intent(in), optional :: label, physical_unit
      type(plot_labels_t), intent(in), optional :: plot_labels
      type(analysis_object_t), pointer :: obj
```

```
      type(histogram_t), pointer :: h
      call analysis_store_init_object (id, AN_HISTOGRAM, obj)
      h => analysis_object_get_histogram_ptr (obj)
      call histogram_init (h, &
           lower_bound, upper_bound, bin_width, &
           label, physical_unit, plot_labels)
    end subroutine analysis_init_histogram_bin_width
```

⟨*Analysis: public*⟩+≡
```
    public :: analysis_init_plot
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_init_plot (id, lower_bound, upper_bound, plot_labels)
      type(string_t), intent(in) :: id
      real(default), intent(in) :: lower_bound, upper_bound
      type(plot_labels_t), intent(in), optional :: plot_labels
      type(analysis_object_t), pointer :: obj
      type(plot_t), pointer :: plot
      call analysis_store_init_object (id, AN_PLOT, obj)
      plot => analysis_object_get_plot_ptr (obj)
      call plot_init (plot, &
           lower_bound, upper_bound, plot_labels)
    end subroutine analysis_init_plot
```

Recording data

This procedure resets an object or the whole store to its initial state.

⟨*Analysis: public*⟩+≡
```
    public :: analysis_clear
```

⟨*Analysis: interfaces*⟩+≡
```
    interface analysis_clear
       module procedure analysis_store_clear_obj
       module procedure analysis_store_clear_all
    end interface
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_store_clear_obj (id)
      type(string_t), intent(in) :: id
      type(analysis_object_t), pointer :: obj
      obj => analysis_store_get_object_ptr (id)
      if (associated (obj)) then
         call analysis_object_clear (obj)
      end if
    end subroutine analysis_store_clear_obj

    subroutine analysis_store_clear_all ()
      type(analysis_object_t), pointer :: obj
      obj => analysis_store%first
      do while (associated (obj))
         call analysis_object_clear (obj)
         obj => obj%next
      end do
```

```
    end subroutine analysis_store_clear_all
```

There is one generic recording function whose behavior depends on the type of analysis object.

⟨*Analysis: public*⟩+≡
```
    public :: analysis_record_data
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_record_data (id, x, y, xerr, yerr, &
        weight, excess, success, exist)
      type(string_t), intent(in) :: id
      real(default), intent(in) :: x
      real(default), intent(in), optional :: y, xerr, yerr, weight, excess
      logical, intent(out), optional :: success, exist
      type(analysis_object_t), pointer :: obj
      obj => analysis_store_get_object_ptr (id)
      if (associated (obj)) then
         call analysis_object_record_data (obj, x, y, xerr, yerr, &
              weight, excess, success)
         if (present (exist))  exist = .true.
      else
         if (present (success))  success = .false.
         if (present (exist))  exist = .false.
      end if
    end subroutine analysis_record_data
```

Retrieve generic results

The following functions should work for all kinds of analysis object:

⟨*Analysis: public*⟩+≡
```
    public :: analysis_get_n_entries
    public :: analysis_get_average
    public :: analysis_get_error
```

⟨*Analysis: procedures*⟩+≡
```
    function analysis_get_n_entries (id, within_bounds) result (n)
      integer :: n
      type(string_t), intent(in) :: id
      logical, intent(in), optional :: within_bounds
      type(analysis_object_t), pointer :: obj
      obj => analysis_store_get_object_ptr (id)
      if (associated (obj)) then
         n = analysis_object_get_n_entries (obj, within_bounds)
      else
         n = 0
      end if
    end function analysis_get_n_entries

    function analysis_get_average (id, within_bounds) result (avg)
      real(default) :: avg
      type(string_t), intent(in) :: id
      type(analysis_object_t), pointer :: obj
      logical, intent(in), optional :: within_bounds
```

```
      obj => analysis_store_get_object_ptr (id)
      if (associated (obj)) then
         avg = analysis_object_get_average (obj, within_bounds)
      else
         avg = 0
      end if
    end function analysis_get_average

    function analysis_get_error (id, within_bounds) result (err)
      real(default) :: err
      type(string_t), intent(in) :: id
      type(analysis_object_t), pointer :: obj
      logical, intent(in), optional :: within_bounds
      obj => analysis_store_get_object_ptr (id)
      if (associated (obj)) then
         err = analysis_object_get_error (obj, within_bounds)
      else
         err = 0
      end if
    end function analysis_get_error
```

Return true if any analysis object is graphical

⟨*Analysis: public*⟩+≡
```
    public :: analysis_has_plots
```

⟨*Analysis: interfaces*⟩+≡
```
    interface analysis_has_plots
       module procedure analysis_has_plots_any
       module procedure analysis_has_plots_obj
    end interface
```

⟨*Analysis: procedures*⟩+≡
```
    function analysis_has_plots_any () result (flag)
      logical :: flag
      type(analysis_object_t), pointer :: obj
      flag = .false.
      obj => analysis_store%first
      do while (associated (obj))
         flag = analysis_object_has_plot (obj)
         if (flag)  return
      end do
    end function analysis_has_plots_any

    function analysis_has_plots_obj (id) result (flag)
      logical :: flag
      type(string_t), dimension(:), intent(in) :: id
      type(analysis_object_t), pointer :: obj
      integer :: i
      flag = .false.
      do i = 1, size (id)
         obj => analysis_store_get_object_ptr (id(i))
         if (associated (obj)) then
            flag = analysis_object_has_plot (obj)
            if (flag)  return
```

```
      end if
    end do
  end function analysis_has_plots_obj
```

Output

⟨*Analysis: public*⟩+≡
```
    public :: analysis_write
```

⟨*Analysis: interfaces*⟩+≡
```
    interface analysis_write
       module procedure analysis_write_object
       module procedure analysis_write_all
    end interface
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_write_object (id, unit)
      type(string_t), intent(in) :: id
      integer, intent(in), optional :: unit
      type(analysis_object_t), pointer :: obj
      obj => analysis_store_get_object_ptr (id)
      if (associated (obj)) then
         call analysis_object_write (obj, unit)
      else
         call msg_error ("Analysis object '" // char (id) // "' not found")
      end if
    end subroutine analysis_write_object

    subroutine analysis_write_all (unit)
      integer, intent(in), optional :: unit
      type(analysis_object_t), pointer :: obj
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      obj => analysis_store%first
      do while (associated (obj))
         call analysis_object_write (obj, unit)
         obj => obj%next
      end do
    end subroutine analysis_write_all
```

⟨*Analysis: public*⟩+≡
```
    public :: analysis_write_driver
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_write_driver (filename_data, id, unit)
      type(string_t), intent(in) :: filename_data
      type(string_t), dimension(:), intent(in), optional :: id
      integer, intent(in), optional :: unit
      if (present (id)) then
         call analysis_store_write_driver_obj (filename_data, id, unit)
      else
         call analysis_store_write_driver_all (filename_data, unit)
      end if
```

```
        end subroutine analysis_write_driver
```

⟨*Analysis: public*⟩+≡
```
    public :: analysis_compile_tex
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_compile_tex (file, has_gmlcode, os_data)
      type(string_t), intent(in) :: file
      logical, intent(in) :: has_gmlcode
      type(os_data_t), intent(in) :: os_data
      type(string_t) :: setenv
      integer :: status
      if (os_data%event_analysis_ps) then
         BLOCK: do
            if (os_data%whizard_texpath /= "") then
               setenv = "TEXINPUTS=" // os_data%whizard_texpath // ":$TEXINPUTS "
            else
               setenv = ""
            end if
            call os_system_call (setenv // os_data%latex // " " // file, status)
            if (status /= 0)  exit BLOCK
            if (has_gmlcode) then
               call os_system_call (os_data%gml // " " // file, status)
               if (status /= 0)  exit BLOCK
               call os_system_call (setenv // os_data%latex // " " // file, &
                    status)
               if (status /= 0)  exit BLOCK
            end if
            call os_system_call (os_data%dvips // " " // file, status)
            if (status /= 0)  exit BLOCK
            if (os_data%event_analysis_pdf) then
               call os_system_call (os_data%ps2pdf // " " // file // ".ps", &
                                    status)
               if (status /= 0)  exit BLOCK
            end if
            exit BLOCK
         end do BLOCK
         if (status /= 0) then
            call msg_error ("Unable to compile analysis output file")
         end if
      end if
    end subroutine analysis_compile_tex
```

5.1.11   Test

⟨*Analysis: public*⟩+≡
```
    public :: analysis_test
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_test ()
      call analysis_test1 ()
      call analysis_final ()
    end subroutine analysis_test
```

⟨*Analysis: procedures*⟩+≡

```
    subroutine analysis_test1 ()
      type(string_t) :: id1, id2, id3, id4
      integer :: i
      id1 = "foo"
      id2 = "bar"
      id3 = "hist"
      id4 = "plot"
      call analysis_init_observable (id1)
      call analysis_init_observable (id2)
      call analysis_init_histogram_bin_width &
           (id3, 0.5_default, 5.5_default, 1._default)
      call analysis_init_plot (id4, 0.5_default, 5.5_default)
      do i = 1, 3
         print *, "data = ", real(i,default)
         call analysis_record_data (id1, real(i,default))
         call analysis_record_data (id2, real(i,default), &
                                       weight=real(i,default))
         call analysis_record_data (id3, real(i,default))
         call analysis_record_data (id4, real(i,default), real(i,default)**2)
      end do
  1   format (A,10(1x,I5))
  2   format (A,10(1x,F5.3))
      print 1, "n_entries = ", &
           analysis_get_n_entries (id1), &
           analysis_get_n_entries (id2), &
           analysis_get_n_entries (id3), &
           analysis_get_n_entries (id3, within_bounds = .true.), &
           analysis_get_n_entries (id4), &
           analysis_get_n_entries (id4, within_bounds = .true.)
      print 2, "average   = ", &
           analysis_get_average (id1), &
           analysis_get_average (id2), &
           analysis_get_average (id3), &
           analysis_get_average (id3, within_bounds = .true.)
      print 2, "error     = ", &
           analysis_get_error (id1), &
           analysis_get_error (id2), &
           analysis_get_error (id3), &
           analysis_get_error (id3, within_bounds = .true.)
      print *, "clear #2"
      call analysis_clear (id2)
      do i = 4, 6
         print *, "data = ", real(i,default)
         call analysis_record_data (id1, real(i,default))
         call analysis_record_data (id2, real(i,default), &
                                       weight=real(i,default))
         call analysis_record_data (id3, real(i,default))
         call analysis_record_data (id4, real(i,default), real(i,default)**2)
      end do
      print 1, "n_entries = ", &
           analysis_get_n_entries (id1), &
           analysis_get_n_entries (id2), &
           analysis_get_n_entries (id3), &
```

```
         analysis_get_n_entries (id3, within_bounds = .true.), &
         analysis_get_n_entries (id4), &
         analysis_get_n_entries (id4, within_bounds = .true.)
    print 2, "average   = ", &
         analysis_get_average (id1), &
         analysis_get_average (id2), &
         analysis_get_average (id3), &
         analysis_get_average (id3, within_bounds = .true.)
    print 2, "error     = ", &
         analysis_get_error (id1), &
         analysis_get_error (id2), &
         analysis_get_error (id3), &
         analysis_get_error (id3, within_bounds = .true.)
    print *
    call analysis_write ()
    call analysis_clear ()
end subroutine analysis_test1
```

5.2 PDG arrays

For defining aliases, we introduce a special type which holds a set of (integer)
PDG codes.

⟨pdg_arrays.f90⟩≡
  ⟨*File header*⟩

  ```
module pdg_arrays
```

  ⟨*Use file utils*⟩

  ⟨*Standard module head*⟩

  ⟨*PDG arrays: public*⟩

  ⟨*PDG arrays: parameters*⟩

  ⟨*PDG arrays: types*⟩

  ⟨*PDG arrays: interfaces*⟩

  ```
contains
```

  ⟨*PDG arrays: procedures*⟩

  ```
end module pdg_arrays
```

5.2.1 Type definition

Using an allocatable array eliminates the need for initializer and/or finalizer.

⟨*PDG arrays: public*⟩≡
```
    public :: pdg_array_t
```

⟨*PDG arrays: types*⟩≡
```
    type :: pdg_array_t
       private
       integer, dimension(:), allocatable :: pdg
    end type pdg_array_t
```

Output

⟨*PDG arrays: public*⟩+≡
```
    public :: pdg_array_write
```

⟨*PDG arrays: procedures*⟩≡
```
    subroutine pdg_array_write (aval, unit)
      type(pdg_array_t), intent(in) :: aval
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)", advance="no")  "PDG("
      if (allocated (aval%pdg)) then
         do i = 1, size (aval%pdg)
            if (i > 1)  write (u, "(A)", advance="no")  ", "
```

```
            write (u, "(I0)", advance="no")  aval%pdg(i)
        end do
    end if
    write (u, "(A)", advance="no")  ")"
end subroutine pdg_array_write
```

5.2.2 Parameters

We need an UNDEFINED value:

⟨*PDG arrays: parameters*⟩≡
```
    integer, parameter, public :: UNDEFINED = 0
```

5.2.3 Basic operations

Assignment. We define assignment from and to an integer array. Note that the integer array, if it is the l.h.s., must be declared allocatable by the caller.

⟨*PDG arrays: public*⟩+≡
```
    public :: assignment(=)
```

⟨*PDG arrays: interfaces*⟩≡
```
    interface assignment(=)
       module procedure pdg_array_from_int_array
       module procedure pdg_array_from_int
       module procedure int_array_from_pdg_array
    end interface
```

⟨*PDG arrays: procedures*⟩+≡
```
    subroutine pdg_array_from_int_array (aval, iarray)
      type(pdg_array_t), intent(out) :: aval
      integer, dimension(:), intent(in) :: iarray
      allocate (aval%pdg (size (iarray)))
      aval%pdg = iarray
    end subroutine pdg_array_from_int_array

    elemental subroutine pdg_array_from_int (aval, int)
      type(pdg_array_t), intent(out) :: aval
      integer, intent(in) :: int
      allocate (aval%pdg (1))
      aval%pdg = int
    end subroutine pdg_array_from_int

    subroutine int_array_from_pdg_array (iarray, aval)
      integer, dimension(:), allocatable, intent(out) :: iarray
      type(pdg_array_t), intent(in) :: aval
      if (allocated (aval%pdg)) then
         allocate (iarray (size (aval%pdg)))
         iarray = aval%pdg
      else
         allocate (iarray (0))
      end if
    end subroutine int_array_from_pdg_array
```

The only nontrivial operation: concatenate two PDG arrays

⟨*PDG arrays: public*⟩+≡
```
public :: operator(//)
```

⟨*PDG arrays: interfaces*⟩+≡
```
interface operator(//)
   module procedure concat_pdg_arrays
end interface
```

⟨*PDG arrays: procedures*⟩+≡
```
function concat_pdg_arrays (aval1, aval2) result (aval)
  type(pdg_array_t) :: aval
  type(pdg_array_t), intent(in) :: aval1, aval2
  integer :: n1, n2
  if (allocated (aval1%pdg) .and. allocated (aval2%pdg)) then
     n1 = size (aval1%pdg)
     n2 = size (aval2%pdg)
     allocate (aval%pdg (n1 + n2))
     aval%pdg(:n1) = aval1%pdg
     aval%pdg(n1+1:) = aval2%pdg
  else if (allocated (aval1%pdg)) then
     aval = aval1
  else if (allocated (aval2%pdg)) then
     aval = aval2
  end if
end function concat_pdg_arrays
```

5.2.4   Matching

A PDG array matches a given PDG code if the code is present within the array.
If either one is zero (UNDEFINED), the match also succeeds.

⟨*PDG arrays: public*⟩+≡
```
public :: operator(.match.)
```

⟨*PDG arrays: interfaces*⟩+≡
```
interface operator(.match.)
   module procedure pdg_array_match_integer
end interface
```

⟨*PDG arrays: procedures*⟩+≡
```
elemental function pdg_array_match_integer (aval, pdg) result (flag)
  logical :: flag
  type(pdg_array_t), intent(in) :: aval
  integer, intent(in) :: pdg
  if (allocated (aval%pdg)) then
     flag = pdg == UNDEFINED &
          .or. any (aval%pdg == UNDEFINED) &
          .or. any (aval%pdg == pdg)
  else
     flag = .false.
  end if
end function pdg_array_match_integer
```

Equivalence. Two PDG arrays are equivalent if either both contain `UNDEFINED` or each element of array 1 is present in array 2, and vice versa.

⟨*PDG arrays: public*⟩+≡
```
public :: operator(.eqv.)
public :: operator(.neqv.)
```

⟨*PDG arrays: interfaces*⟩+≡
```
interface operator(.eqv.)
   module procedure pdg_array_equivalent
end interface
interface operator(.neqv.)
   module procedure pdg_array_inequivalent
end interface
```

⟨*PDG arrays: procedures*⟩+≡
```
function pdg_array_equivalent (aval1, aval2) result (eq)
  logical :: eq
  type(pdg_array_t), intent(in) :: aval1, aval2
  logical, dimension(:), allocatable :: match1, match2
  integer :: i
  if (allocated (aval1%pdg) .and. allocated (aval2%pdg)) then
     eq = any (aval1%pdg == UNDEFINED) &
          .and. any (aval1%pdg == UNDEFINED)
     if (.not. eq) then
        allocate (match1 (size (aval1%pdg)))
        allocate (match2 (size (aval2%pdg)))
        match1 = .false.
        match2 = .false.
        do i = 1, size (aval1%pdg)
           match2 = match2 .or. aval1%pdg(i) == aval2%pdg
        end do
        do i = 1, size (aval2%pdg)
           match1 = match1 .or. aval2%pdg(i) == aval1%pdg
        end do
        eq = all (match1) .and. all (match2)
     end if
  else
     eq = .false.
  end if
end function pdg_array_equivalent

function pdg_array_inequivalent (aval1, aval2) result (neq)
  logical :: neq
  type(pdg_array_t), intent(in) :: aval1, aval2
  neq = .not. pdg_array_equivalent (aval1, aval2)
end function pdg_array_inequivalent
```

5.3 Particle lists

Within expressions (see below), we need a representation for particle lists, which is independent from other particle representations within WHIZARD. This is provided here.

⟨prt_lists.f90⟩≡
  ⟨*File header*⟩

  module prt_lists

  ⟨*Use kinds*⟩
  ⟨*Use file utils*⟩
    use lorentz !NODEP!
    use sorting
    use pdg_arrays

  ⟨*Standard module head*⟩

  ⟨*Prt lists: public*⟩

  ⟨*Prt lists: parameters*⟩

  ⟨*Prt lists: types*⟩

  ⟨*Prt lists: interfaces*⟩

  contains

  ⟨*Prt lists: procedures*⟩

  end module prt_lists

5.3.1 Particles

For the purpose of this module, a particle has a type which can indicate an incoming, outgoing, or composite particle, flavor and helicity codes (integer, undefined for composite), four-momentum and invariant mass squared. Furthermore, each particle has an allocatable array of ancestors – particle indices which indicate the building blocks of a composite particle. For an incoming/outgoing particle, the array contains only the index of the particle itself.

For incoming particles, the momentum is inverted before storing it in the particle object.

⟨*Prt lists: parameters*⟩≡
```
    integer, parameter, public :: PRT_UNDEFINED = 0
    integer, parameter, public :: PRT_BEAM = -9
    integer, parameter, public :: PRT_INCOMING = 1
    integer, parameter, public :: PRT_OUTGOING = 2
    integer, parameter, public :: PRT_COMPOSITE = 3
    integer, parameter, public :: PRT_VIRTUAL = 4
    integer, parameter, public :: PRT_RESONANT = 5
    integer, parameter, public :: PRT_BEAM_REMNANT = 9
```

The type

We initialize only the type here and mark as unpolarized. The initializers below do the rest.

⟨*Prt lists: public*⟩≡
```
public :: prt_t
```

⟨*Prt lists: types*⟩≡
```
type :: prt_t
   private
   integer :: type = PRT_UNDEFINED
   integer :: pdg
   logical :: polarized = .false.
   integer :: h
   type(vector4_t) :: p
   real(default) :: p2
   integer, dimension(:), allocatable :: src
end type prt_t
```

Initializers. Polarization is set separately. Finalizers are not needed.

⟨*Prt lists: procedures*⟩≡
```
subroutine prt_init_incoming (prt, pdg, p, p2, src)
  type(prt_t), intent(out) :: prt
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in) :: src
  prt%type = PRT_INCOMING
  call prt_set (prt, pdg, - p, p2, src)
end subroutine prt_init_incoming

subroutine prt_init_outgoing (prt, pdg, p, p2, src)
  type(prt_t), intent(out) :: prt
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in) :: src
  prt%type = PRT_OUTGOING
  call prt_set (prt, pdg, p, p2, src)
end subroutine prt_init_outgoing

subroutine prt_init_composite (prt, p, src)
  type(prt_t), intent(out) :: prt
  type(vector4_t), intent(in) :: p
  integer, dimension(:), intent(in) :: src
  prt%type = PRT_COMPOSITE
  call prt_set (prt, 0, p, p**2, src)
end subroutine prt_init_composite
```

This version is for temporary particle objects, so the `src` array is not set.

⟨*Prt lists: public*⟩+≡
```
public :: prt_init_combine
```

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_init_combine (prt, prt1, prt2)
      type(prt_t), intent(out) :: prt
      type(prt_t), intent(in) :: prt1, prt2
      type(vector4_t) :: p
      integer, dimension(0) :: src
      prt%type = PRT_COMPOSITE
      p = prt1%p + prt2%p
      call prt_set (prt, 0, p, p**2, src)
    end subroutine prt_init_combine
```

Accessing contents

⟨*Prt lists: public*⟩+≡
```
    public :: prt_get_pdg
```
⟨*Prt lists: procedures*⟩+≡
```
    elemental function prt_get_pdg (prt) result (pdg)
      integer :: pdg
      type(prt_t), intent(in) :: prt
      pdg = prt%pdg
    end function prt_get_pdg
```

⟨*Prt lists: public*⟩+≡
```
    public :: prt_get_momentum
```
⟨*Prt lists: procedures*⟩+≡
```
    elemental function prt_get_momentum (prt) result (p)
      type(vector4_t) :: p
      type(prt_t), intent(in) :: prt
      p = prt%p
    end function prt_get_momentum
```

⟨*Prt lists: public*⟩+≡
```
    public :: prt_get_msq
```
⟨*Prt lists: procedures*⟩+≡
```
    elemental function prt_get_msq (prt) result (msq)
      real(default) :: msq
      type(prt_t), intent(in) :: prt
      msq = prt%p2
    end function prt_get_msq
```

⟨*Prt lists: public*⟩+≡
```
    public :: prt_is_polarized
```
⟨*Prt lists: procedures*⟩+≡
```
    elemental function prt_is_polarized (prt) result (flag)
      logical :: flag
      type(prt_t), intent(in) :: prt
      flag = prt%polarized
    end function prt_is_polarized
```

⟨*Prt lists: public*⟩+≡
```
    public :: prt_get_helicity
```

⟨*Prt lists: procedures*⟩+≡
```
    elemental function prt_get_helicity (prt) result (h)
      integer :: h
      type(prt_t), intent(in) :: prt
      h = prt%h
    end function prt_get_helicity
```

Setting data

Set the PDG, momentum and momentum squared, and ancestors. If allocate-on-assignment is available, this can be simplified.

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_set (prt, pdg, p, p2, src)
      type(prt_t), intent(inout) :: prt
      integer, intent(in) :: pdg
      type(vector4_t), intent(in) :: p
      real(default), intent(in) :: p2
      integer, dimension(:), intent(in) :: src
      prt%pdg = pdg
      prt%p = p
      prt%p2 = p2
      if (allocated (prt%src)) then
         if (size (src) /= size (prt%src)) then
            deallocate (prt%src)
            allocate (prt%src (size (src)))
         end if
      else
         allocate (prt%src (size (src)))
      end if
      prt%src = src
    end subroutine prt_set
```

Set the momentum separately.

⟨*Prt lists: procedures*⟩+≡
```
    elemental subroutine prt_set_p (prt, p)
      type(prt_t), intent(inout) :: prt
      type(vector4_t), intent(in) :: p
      prt%p = p
    end subroutine prt_set_p
```

Set helicity (optional).

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_polarize (prt, h)
      type(prt_t), intent(inout) :: prt
      integer, intent(in) :: h
      prt%polarized = .true.
      prt%h = h
    end subroutine prt_polarize
```

Output

⟨*Prt lists: public*⟩+≡
```
    public :: prt_write
```

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_write (prt, unit)
      type(prt_t), intent(in) :: prt
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,A)", advance="no")  "prt("
      select case (prt%type)
      case (PRT_UNDEFINED);    write (u, "('?')", advance="no")
      case (PRT_INCOMING);     write (u, "('i:')", advance="no")
      case (PRT_OUTGOING);     write (u, "('o:')", advance="no")
      case (PRT_COMPOSITE);    write (u, "('c:')", advance="no")
      end select
      select case (prt%type)
      case (PRT_INCOMING, PRT_OUTGOING)
         if (prt%polarized) then
            write (u, "(I0,'/',I0,'|')", advance="no")  prt%pdg, prt%h
         else
            write (u, "(I0,'|')", advance="no") prt%pdg
         end if
      end select
      select case (prt%type)
      case (PRT_INCOMING, PRT_OUTGOING, PRT_COMPOSITE)
         write (u, "(1PE12.5,';',1PE12.5,',',1PE12.5,',',1PE12.5)", advance="no") &
               array_from_vector4 (prt%p)
         write (u, "('|',1PE12.5)", advance="no")  prt%p2
      end select
      if (allocated (prt%src)) then
         write (u, "('|')", advance="no")
         do i = 1, size (prt%src)
            write (u, "(1x,I0)", advance="no")  prt%src(i)
         end do
      end if
      write (u, "(A)")  ")"
    end subroutine prt_write
```

Tools

Two particles match if their `src` arrays are the same.

⟨*Prt lists: interfaces*⟩≡
```
    interface operator(.match.)
       module procedure prt_match
    end interface
```

⟨*Prt lists: procedures*⟩+≡
```
    elemental function prt_match (prt1, prt2) result (match)
      logical :: match
      type(prt_t), intent(in) :: prt1, prt2
```

```
      if (size (prt1%src) == size (prt2%src)) then
         match = all (prt1%src == prt2%src)
      else
         match = .false.
      end if
   end function prt_match
```

The combine operation makes a pseudoparticle whose momentum is the result of adding (the momenta of) the pair of input particles. We trace the particles from which a particle is built by storing a `src` array. Each particle entry in the `src` list contains a list of indices which indicates its building blocks. The indices refer to an original list of particles. Index lists are sorted, and they contain no element more than once.

We thus require that in a given pseudoparticle, each original particle occurs at most once.

The result is intent(inout), so it will not be initialized when the subroutine is entered.

⟨*Prt lists: procedures*⟩+≡
```
   subroutine prt_combine (prt, prt_in1, prt_in2, ok)
     type(prt_t), intent(inout) :: prt
     type(prt_t), intent(in) :: prt_in1, prt_in2
     logical :: ok
     integer, dimension(:), allocatable :: src
     call combine_index_lists (src, prt_in1%src, prt_in2%src)
     ok = allocated (src)
     if (ok)  call prt_init_composite (prt, prt_in1%p + prt_in2%p, src)
   end subroutine prt_combine
```

This variant does not produce the combined particle, it just checks whether the combination is valid (no common `src` entry).

⟨*Prt lists: public*⟩+≡
```
   public :: are_disjoint
```

⟨*Prt lists: procedures*⟩+≡
```
   function are_disjoint (prt_in1, prt_in2) result (flag)
     logical :: flag
     type(prt_t), intent(in) :: prt_in1, prt_in2
     flag = index_lists_are_disjoint (prt_in1%src, prt_in2%src)
   end function are_disjoint
```

`src` Lists with length > 1 are built by a `combine` operation which merges the lists in a sorted manner. If the result would have a duplicate entry, it is discarded, and the result is unallocated.

⟨*Prt lists: procedures*⟩+≡
```
   subroutine combine_index_lists (res, src1, src2)
     integer, dimension(:), intent(in) :: src1, src2
     integer, dimension(:), allocatable :: res
     integer :: i1, i2, i
     allocate (res (size (src1) + size (src2)))
     i1 = 1
     i2 = 1
```

```
      LOOP: do i = 1, size (res)
         if (src1(i1) < src2(i2)) then
            res(i) = src1(i1);   i1 = i1 + 1
            if (i1 > size (src1)) then
               res(i+1:) = src2(i2:)
               exit LOOP
            end if
         else if (src1(i1) > src2(i2)) then
            res(i) = src2(i2);   i2 = i2 + 1
            if (i2 > size (src2)) then
               res(i+1:) = src1(i1:)
               exit LOOP
            end if
         else
            deallocate (res)
            exit LOOP
         end if
      end do LOOP
   end subroutine combine_index_lists
```

This function is similar, but it does not actually merge the list, it just checks whether they are disjoint (no common `src` entry).

⟨*Prt lists: procedures*⟩+≡
```
   function index_lists_are_disjoint (src1, src2) result (flag)
     logical :: flag
     integer, dimension(:), intent(in) :: src1, src2
     integer :: i1, i2, i
     flag = .true.
     i1 = 1
     i2 = 1
     LOOP: do i = 1, size (src1) + size (src2)
        if (src1(i1) < src2(i2)) then
           i1 = i1 + 1
           if (i1 > size (src1)) then
              exit LOOP
           end if
        else if (src1(i1) > src2(i2)) then
           i2 = i2 + 1
           if (i2 > size (src2)) then
              exit LOOP
           end if
        else
           flag = .false.
           exit LOOP
        end if
     end do LOOP
   end function index_lists_are_disjoint
```

5.3.2   Particle lists

Particles are usually collected in lists. The list is actually implemented as a dynamically allocated array, which need not be completely filled.

Type definition

⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_t
```

⟨*Prt lists: types*⟩+≡
```
    type :: prt_list_t
      private
      integer :: n = 0
      type(prt_t), dimension(:), allocatable :: prt
    end type prt_list_t
```

Initialize, allocating with size zero (default) or given size. The number of contained particles is set equal to the size.

⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_init
```

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_list_init (prt_list, n)
      type(prt_list_t), intent(out) :: prt_list
      integer, intent(in), optional :: n
      if (present (n)) then
        allocate (prt_list%prt (n))
        prt_list%n = n
      else
        allocate (prt_list%prt (0))
      end if
    end subroutine prt_list_init
```

(Re-)allocate the particle list with some given size. If the size is greater than the previous one, do a real reallocation. Otherwise, just reset the recorded size. Contents are untouched, but become invalid.

⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_reset
```

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_list_reset (prt_list, n)
      type(prt_list_t), intent(inout) :: prt_list
      integer, intent(in) :: n
      if (n > size (prt_list%prt)) then
        deallocate (prt_list%prt)
        allocate (prt_list%prt (n))
      end if
      prt_list%n = n
    end subroutine prt_list_reset
```

Output.

⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_write
```

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_list_write (prt_list, unit, prefix)
      type(prt_list_t), intent(in) :: prt_list
      integer, intent(in), optional :: unit
```

```
    character(*), intent(in), optional :: prefix
    integer :: u, i
    u = output_unit (unit);  if (u < 0)  return
    write (u, *) "Particle list:"
    do i = 1, prt_list%n
       if (present (prefix))  write (u, "(A)", advance="no") prefix
       write (u, "(1x,I0)", advance="no")  i
       call prt_write (prt_list%prt(i), unit)
    end do
  end subroutine prt_list_write
```

Defined assignment: transfer only meaningful entries. This is a deep copy (as would be default assignment).

⟨*Prt lists: interfaces*⟩+≡
```
    interface assignment(=)
       module procedure prt_list_assign
    end interface
```

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_list_assign (prt_list, prt_list_in)
      type(prt_list_t), intent(inout) :: prt_list
      type(prt_list_t), intent(in) :: prt_list_in
      integer :: i
      if (.not. allocated (prt_list%prt))  call prt_list_init (prt_list)
      call prt_list_reset (prt_list, prt_list_in%n)
      do i = 1, prt_list%n
         prt_list%prt(i) = prt_list_in%prt(i)
      end do
    end subroutine prt_list_assign
```

Fill contents

Store incoming/outgoing particles which are completely defined.

⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_set_incoming
    public :: prt_list_set_outgoing
    public :: prt_list_set_composite
```

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_list_set_incoming (prt_list, i, pdg, p, p2, src)
      type(prt_list_t), intent(inout) :: prt_list
      integer, intent(in) :: i
      integer, intent(in) :: pdg
      type(vector4_t), intent(in) :: p
      real(default), intent(in) :: p2
      integer, dimension(:), intent(in), optional :: src
      if (present (src)) then
         call prt_init_incoming (prt_list%prt(i), pdg, p, p2, src)
      else
         call prt_init_incoming (prt_list%prt(i), pdg, p, p2, (/ i /))
      end if
    end subroutine prt_list_set_incoming
```

276

```
subroutine prt_list_set_outgoing (prt_list, i, pdg, p, p2, src)
  type(prt_list_t), intent(inout) :: prt_list
  integer, intent(in) :: i
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in), optional :: src
  if (present (src)) then
     call prt_init_outgoing (prt_list%prt(i), pdg, p, p2, src)
  else
     call prt_init_outgoing (prt_list%prt(i), pdg, p, p2, (/ i /))
  end if
end subroutine prt_list_set_outgoing

subroutine prt_list_set_composite (prt_list, i, p, src)
  type(prt_list_t), intent(inout) :: prt_list
  integer, intent(in) :: i
  type(vector4_t), intent(in) :: p
  integer, dimension(:), intent(in) :: src
  call prt_init_composite (prt_list%prt(i), p, src)
end subroutine prt_list_set_composite
```

Separately assign momentum, simultaneously for all incoming/outgoing particles. We assume that the list length coincides with the array size.

⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_set_p_incoming
    public :: prt_list_set_p_outgoing
```

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_list_set_p_incoming (prt_list, p)
      type(prt_list_t), intent(inout) :: prt_list
      type(vector4_t), dimension(:), intent(in) :: p
      integer :: n_in
      n_in = size (p)
      call prt_set_p (prt_list%prt(:n_in), p)
    end subroutine prt_list_set_p_incoming

    subroutine prt_list_set_p_outgoing (prt_list, p)
      type(prt_list_t), intent(inout) :: prt_list
      type(vector4_t), dimension(:), intent(in) :: p
      integer :: n_in, n_out
      n_out = size (p)
      n_in = prt_list%n - n_out
      call prt_set_p (prt_list%prt(n_in+1:), p)
    end subroutine prt_list_set_p_outgoing
```

Set polarization for an entry

⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_polarize
```

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_list_polarize (prt_list, i, h)
```

```
      type(prt_list_t), intent(inout) :: prt_list
      integer, intent(in) :: i, h
      call prt_polarize (prt_list%prt(i), h)
    end subroutine prt_list_polarize
```

Accessing contents

Return true if the particle list has entries.
⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_is_nonempty
```

⟨*Prt lists: procedures*⟩+≡
```
    function prt_list_is_nonempty (prt_list) result (flag)
      logical :: flag
      type(prt_list_t), intent(in) :: prt_list
      flag = prt_list%n /= 0
    end function prt_list_is_nonempty
```

Return the number of entries
⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_get_length
```

⟨*Prt lists: procedures*⟩+≡
```
    function prt_list_get_length (prt_list) result (length)
      integer :: length
      type(prt_list_t), intent(in) :: prt_list
      length = prt_list%n
    end function prt_list_get_length
```

Return a specific particle. The index is not checked for validity.
⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_get_prt
```

⟨*Prt lists: procedures*⟩+≡
```
    function prt_list_get_prt (prt_list, i) result (prt)
      type(prt_t) :: prt
      type(prt_list_t), intent(in) :: prt_list
      integer, intent(in) :: i
      prt = prt_list%prt(i)
    end function prt_list_get_prt
```

Operations with particle lists

The join operation joins two particle lists. When appending the elements of the second list, we check for each particle whether it is already in the first list. If yes, it is discarded. The result list should be initialized already.

If a mask is present, it refers to the second particle list. Particles where the mask is not set are discarded.
⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_join
```

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_list_join (prt_list, pl1, pl2, mask2)
      type(prt_list_t), intent(inout) :: prt_list
      type(prt_list_t), intent(in) :: pl1, pl2
      logical, dimension(:), intent(in) :: mask2
      integer :: n1, n2, i, n
      n1 = pl1%n
      n2 = pl2%n
      call prt_list_reset (prt_list, n1 + n2)
      prt_list%prt(:n1)   = pl1%prt(:n1)
      n = n1
      do i = 1, pl2%n
         if (mask2(i) .and. .not. any (pl2%prt(i) .match. pl1%prt(:pl1%n))) then
            n = n + 1
            prt_list%prt(n) = pl2%prt(i)
         end if
      end do
      prt_list%n = n
    end subroutine prt_list_join
```

The combine operation makes a particle list whose entries are the result of
adding (the momenta of) each pair of particles in the input lists. We trace the
particles from which a particles is built by storing a `src` array. Each particle
entry in the `src` list contains a list of indices which indicates its building blocks.
The indices refer to an original list of particles. Index lists are sorted, and they
contain no element more than once.

   We thus require that in a given pseudoparticle, each original particle occurs
at most once.

⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_combine
```

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_list_combine (prt_list, pl1, pl2, mask12)
      type(prt_list_t), intent(inout) :: prt_list
      type(prt_list_t), intent(in) :: pl1, pl2
      logical, dimension(:,:), intent(in), optional :: mask12
      integer :: n1, n2, i1, i2, n, j
      logical :: ok
      n1 = pl1%n
      n2 = pl2%n
      call prt_list_reset (prt_list, n1 * n2)
      n = 1
      do i1 = 1, n1
         do i2 = 1, n2
            if (present (mask12)) then
               ok = mask12(i1,i2)
            else
               ok = .true.
            end if
            if (ok)  call prt_combine &
                 (prt_list%prt(n), pl1%prt(i1), pl2%prt(i2), ok)
            if (ok) then
               CHECK_DOUBLES: do j = 1, n - 1
```

```
                    if (prt_list%prt(n) .match. prt_list%prt(j)) then
                        ok = .false.;  exit CHECK_DOUBLES
                    end if
                end do CHECK_DOUBLES
                if (ok)  n = n + 1
            end if
        end do
    end do
    prt_list%n = n - 1
end subroutine prt_list_combine
```

The collect operation makes a single-entry particle list which results from combining (the momenta of) all particles in the input list. As above, the result does not contain an original particle more than once; this is checked for each particle when it is collected. Furthermore, each entry has a mask; where the mask is false, the entry is dropped.

(Thus, if the input particles are already composite, there is some chance that the result depends on the order of the input list and is not as expected. This situation should be avoided.)

⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_collect
```

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_list_collect (prt_list, pl1, mask1)
      type(prt_list_t), intent(inout) :: prt_list
      type(prt_list_t), intent(in) :: pl1
      logical, dimension(:), intent(in) :: mask1
      type(prt_t) :: prt
      integer :: i
      logical :: ok
      call prt_list_reset (prt_list, 1)
      prt_list%n = 0
      do i = 1, pl1%n
         if (mask1(i)) then
            if (prt_list%n == 0) then
               prt_list%n = 1
               prt_list%prt(1) = pl1%prt(i)
            else
               call prt_combine (prt, prt_list%prt(1), pl1%prt(i), ok)
               if (ok)  prt_list%prt(1) = prt
            end if
         end if
      end do
    end subroutine prt_list_collect
```

Return a list of all particles for which the mask is true.

⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_select
```

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_list_select (prt_list, pl, mask1)
      type(prt_list_t), intent(inout) :: prt_list
      type(prt_list_t), intent(in) :: pl
```

```
      logical, dimension(:), intent(in) :: mask1
      integer :: i, n
      call prt_list_reset (prt_list, pl%n)
      n = 0
      do i = 1, pl%n
         if (mask1(i)) then
            n = n + 1
            prt_list%prt(n) = pl%prt(i)
         end if
      end do
      prt_list%n = n
    end subroutine prt_list_select
```

Return a particle list which consists of the single particle with specified `index`. If `index` is negative, count from the end. If it is out of bounds, return an empty list.

⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_extract
```

⟨*Prt lists: procedures*⟩+≡
```
    subroutine prt_list_extract (prt_list, pl, index)
      type(prt_list_t), intent(inout) :: prt_list
      type(prt_list_t), intent(in) :: pl
      integer, intent(in) :: index
      if (index > 0) then
         if (index <= pl%n) then
            call prt_list_reset (prt_list, 1)
            prt_list%prt(1) = pl%prt(index)
         else
            call prt_list_reset (prt_list, 0)
         end if
      else if (index < 0) then
         if (abs (index) <= pl%n) then
            call prt_list_reset (prt_list, 1)
            prt_list%prt(1) = pl%prt(pl%n + 1 + index)
         else
            call prt_list_reset (prt_list, 0)
         end if
      else
         call prt_list_reset (prt_list, 0)
      end if
    end subroutine prt_list_extract
```

Return the list of particles sorted according to increasing values of the provided integer or real array. If no array is given, sort by PDG value.

⟨*Prt lists: public*⟩+≡
```
    public :: prt_list_sort
```

⟨*Prt lists: interfaces*⟩+≡
```
    interface prt_list_sort
       module procedure prt_list_sort_pdg
       module procedure prt_list_sort_int
       module procedure prt_list_sort_real
    end interface
```

⟨*Prt lists: procedures*⟩+≡
```
subroutine prt_list_sort_pdg (prt_list, pl)
  type(prt_list_t), intent(inout) :: prt_list
  type(prt_list_t), intent(in) :: pl
  integer :: n
  n = prt_list%n
  call prt_list_sort_int (prt_list, pl, abs (3 * prt_list%prt(:n)%pdg - 1))
end subroutine prt_list_sort_pdg

subroutine prt_list_sort_int (prt_list, pl, ival)
  type(prt_list_t), intent(inout) :: prt_list
  type(prt_list_t), intent(in) :: pl
  integer, dimension(:), intent(in) :: ival
  call prt_list_reset (prt_list, pl%n)
  prt_list%n = pl%n
  prt_list%prt = pl%prt( order (ival) )
end subroutine prt_list_sort_int

subroutine prt_list_sort_real (prt_list, pl, rval)
  type(prt_list_t), intent(inout) :: prt_list
  type(prt_list_t), intent(in) :: pl
  real(default), dimension(:), intent(in) :: rval
  call prt_list_reset (prt_list, pl%n)
  prt_list%n = pl%n
  prt_list%prt = pl%prt( order (rval) )
end subroutine prt_list_sort_real
```

Return the list of particles which have any of the specified PDG codes.

The `pack` command was buggy in some gfortran versions, therefore it is unrolled. The unrolled version may be more efficient, actually.

⟨*Prt lists: public*⟩+≡
```
public :: prt_list_select_pdg_code
```

⟨*Prt lists: procedures*⟩+≡
```
subroutine prt_list_select_pdg_code (prt_list, aval, prt_list_in, prt_type)
  type(prt_list_t), intent(inout) :: prt_list
  type(pdg_array_t), intent(in) :: aval
  type(prt_list_t), intent(in) :: prt_list_in
  integer, intent(in), optional :: prt_type
  integer :: n_tot, n_match
  logical, dimension(:), allocatable :: mask
  integer :: i, j
  n_tot = prt_list_in%n
  allocate (mask (n_tot))
  forall (i = 1:n_tot) &
      mask(i) = aval .match. prt_list_in%prt(i)%pdg
  if (present (prt_type)) &
      mask = mask .and. prt_list_in%prt(:n_tot)%type == prt_type
  n_match = count (mask)
  call prt_list_reset (prt_list, n_match)
!    prt_list%prt(:n_match) = pack (prt_list_in%prt(:n_tot), mask)
  j = 0
```

```
      do i = 1, n_tot
         if (mask(i)) then
            j = j + 1
            prt_list%prt(j) = prt_list_in%prt(i)
         end if
      end do
   end subroutine prt_list_select_pdg_code
```

5.4 Variables

The user interface deals with variables that are handled similar to full-flegded programming languages. The system will add a lot of predefined variables (model parameters, flags, etc.) that are accessible to the user by the same methods.

Variables can be of various type: logical (boolean/flag), integer, real (default precision), particle lists (used in cut expressions), arrays of PDG codes (aliases for particles), strings. Furthermore, in cut expressions we have unary and binary observables, which are used like real parameters but behave like functions.

⟨`variables.f90`⟩≡

  ⟨*File header*⟩

```
  module variables
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use pdg_arrays
    use prt_lists
```

  ⟨*Standard module head*⟩

  ⟨*Variables: public*⟩

  ⟨*Variables: parameters*⟩

  ⟨*Variables: types*⟩

  ⟨*Variables: interfaces*⟩

```
  contains
```

  ⟨*Variables: procedures*⟩

```
  end module variables
```

5.4.1 Variable list entries

The expressions that we are considering here can contain not just constants, but also variables. We collect them in lists which are associated to particular

nodes in the evaluation tree.

The variables itself can be initialized, but they actually cannot be altered. However, we can initialize them with a pointer to some value elsewhere, so the value changes whenever this target is modified.

Observables are also recorded in the variable list. They are not associated to a value pointer, but to a procedure pointer. The procedure takes one or two evaluation nodes as arguments and produces an integer or a real value, so there are four types and associated procedure-pointer types.

Variable (and constant) values can be of one of the following types:

⟨*Variables: parameters*⟩≡
```
integer, parameter, public :: V_NONE = 0, V_LOG = 1, V_INT = 2, V_REAL = 3
integer, parameter, public :: V_CMPLX = 4, V_PTL = 5, V_PDG = 6, V_STR = 7
integer, parameter, public :: V_OBS1_INT = 11, V_OBS2_INT = 12
integer, parameter, public :: V_OBS1_REAL = 21, V_OBS2_REAL = 22
```

The type

This is an entry in the variable list. It can be of any type; in each case only one value is allocated. It may be physically allocated upon creation, in which case `is_allocated` is true, or it may contain just a pointer to a value somewhere else, in which case `is_allocated` is false.

The flag `is_known` is also a pointer which parallels the use of the value pointer.

The pointer `model_var` is relevant for the parameters of physics models. The global variable list contains copies of all independent parameters, which hold a pointer to the actual variable entry in the model record. The first implementation used direct pointers to this record. However, this induces a problem when during compilation of the variable list the model is undefined. This could happen if the model definition is inside a conditional. The current implementation allows for cleanly switching the underlying model when the command list is executed.

⟨*Variables: public*⟩≡
```
public :: var_entry_t
```

⟨*Variables: types*⟩≡
```
type :: var_entry_t
   private
   integer :: type = V_NONE
   type(string_t) :: name
   logical :: is_allocated = .false.
   logical :: is_locked = .false.
   logical :: is_copy = .false.
   logical :: is_intrinsic = .false.
   type(var_entry_t), pointer :: original => null ()
   logical, pointer :: is_known => null ()
   logical,          pointer :: lval => null ()
   integer,          pointer :: ival => null ()
   real(default),    pointer :: rval => null ()
   complex(default), pointer :: cval => null ()
   type(prt_list_t),  pointer :: pval => null ()
   type(pdg_array_t), pointer :: aval => null ()
```

```
      type(string_t),    pointer :: sval => null ()
      procedure(obs_unary_int),   nopass, pointer :: obs1_int  => null ()
      procedure(obs_unary_real),  nopass, pointer :: obs1_real => null ()
      procedure(obs_binary_int),  nopass, pointer :: obs2_int  => null ()
      procedure(obs_binary_real), nopass, pointer :: obs2_real => null ()
      type(prt_t), pointer :: prt1 => null ()
      type(prt_t), pointer :: prt2 => null ()
      type(var_entry_t), pointer :: next => null ()
   end type var_entry_t
```

Interfaces for the observable functions

⟨*Variables: public*⟩+≡
```
    public :: obs_unary_int
    public :: obs_unary_real
    public :: obs_binary_int
    public :: obs_binary_real
```

⟨*Variables: interfaces*⟩≡
```
    abstract interface
       function obs_unary_int (prt1) result (ival)
         import
         integer :: ival
         type(prt_t), intent(in) :: prt1
       end function obs_unary_int
    end interface
    abstract interface
       function obs_unary_real (prt1) result (rval)
         import
         real(default) :: rval
         type(prt_t), intent(in) :: prt1
       end function obs_unary_real
    end interface
    abstract interface
       function obs_binary_int (prt1, prt2) result (ival)
         import
         integer :: ival
         type(prt_t), intent(in) :: prt1, prt2
       end function obs_binary_int
    end interface
    abstract interface
       function obs_binary_real (prt1, prt2) result (rval)
         import
         real(default) :: rval
         type(prt_t), intent(in) :: prt1, prt2
       end function obs_binary_real
    end interface
```

Initialization

Initialize an entry, optionally with a physical value. We also allocate the
is_known flag and set it if the value is set.

⟨Variables: procedures⟩≡

```fortran
    subroutine var_entry_init_log (var, name, lval, intrinsic)
      type(var_entry_t), intent(out) :: var
      type(string_t), intent(in) :: name
      logical, intent(in), optional :: lval
      logical, intent(in), optional :: intrinsic
      var%name = name
      var%type = V_LOG
      allocate (var%lval, var%is_known)
      if (present (lval)) then
         var%lval = lval
         var%is_known = .true.
      else
         var%is_known = .false.
      end if
      if (present (intrinsic))  var%is_intrinsic = intrinsic
      var%is_allocated = .true.
    end subroutine var_entry_init_log

    subroutine var_entry_init_int (var, name, ival, intrinsic)
      type(var_entry_t), intent(out) :: var
      type(string_t), intent(in) :: name
      integer, intent(in), optional :: ival
      logical, intent(in), optional :: intrinsic
      var%name = name
      var%type = V_INT
      allocate (var%ival, var%is_known)
      if (present (ival)) then
         var%ival = ival
         var%is_known = .true.
      else
         var%is_known = .false.
      end if
      if (present (intrinsic))  var%is_intrinsic = intrinsic
      var%is_allocated = .true.
    end subroutine var_entry_init_int

    subroutine var_entry_init_real (var, name, rval, intrinsic)
      type(var_entry_t), intent(out) :: var
      type(string_t), intent(in) :: name
      real(default), intent(in), optional :: rval
      logical, intent(in), optional :: intrinsic
      var%name = name
      var%type = V_REAL
      allocate (var%rval, var%is_known)
      if (present (rval)) then
         var%rval = rval
         var%is_known = .true.
      else
         var%is_known = .false.
      end if
      if (present (intrinsic))  var%is_intrinsic = intrinsic
      var%is_allocated = .true.
    end subroutine var_entry_init_real
```

```
subroutine var_entry_init_cmplx (var, name, cval, intrinsic)
  type(var_entry_t), intent(out) :: var
  type(string_t), intent(in) :: name
  complex(default), intent(in), optional :: cval
  logical, intent(in), optional :: intrinsic
  var%name = name
  var%type = V_CMPLX
  allocate (var%cval, var%is_known)
  if (present (cval)) then
     var%cval = cval
     var%is_known = .true.
  else
     var%is_known = .false.
  end if
  if (present (intrinsic))  var%is_intrinsic = intrinsic
  var%is_allocated = .true.
end subroutine var_entry_init_cmplx

subroutine var_entry_init_prt_list (var, name, pval, intrinsic)
  type(var_entry_t), intent(out) :: var
  type(string_t), intent(in) :: name
  type(prt_list_t), intent(in), optional :: pval
  logical, intent(in), optional :: intrinsic
  var%name = name
  var%type = V_PTL
  allocate (var%pval, var%is_known)
  if (present (pval)) then
     var%pval = pval
     var%is_known = .true.
  else
     var%is_known = .false.
  end if
  if (present (intrinsic))  var%is_intrinsic = intrinsic
  var%is_allocated = .true.
end subroutine var_entry_init_prt_list

subroutine var_entry_init_pdg_array (var, name, aval, intrinsic)
  type(var_entry_t), intent(out) :: var
  type(string_t), intent(in) :: name
  type(pdg_array_t), intent(in), optional :: aval
  logical, intent(in), optional :: intrinsic
  var%name = name
  var%type = V_PDG
  allocate (var%aval, var%is_known)
  if (present (aval)) then
     var%aval = aval
     var%is_known = .true.
  else
     var%is_known = .false.
  end if
  if (present (intrinsic))  var%is_intrinsic = intrinsic
  var%is_allocated = .true.
end subroutine var_entry_init_pdg_array
```

```
subroutine var_entry_init_string (var, name, sval, intrinsic)
  type(var_entry_t), intent(out) :: var
  type(string_t), intent(in) :: name
  type(string_t), intent(in), optional :: sval
  logical, intent(in), optional :: intrinsic
  var%name = name
  var%type = V_STR
  allocate (var%sval, var%is_known)
  if (present (sval)) then
     var%sval = sval
     var%is_known = .true.
  else
     var%is_known = .false.
  end if
  if (present (intrinsic))  var%is_intrinsic = intrinsic
  var%is_allocated = .true.
end subroutine var_entry_init_string
```

Initialize an entry with a pointer to the value and, for numeric/logical values, a pointer to the `is_known` flag.

⟨*Variables: public*⟩+≡

```
public :: var_entry_init_log_ptr
public :: var_entry_init_int_ptr
public :: var_entry_init_real_ptr
public :: var_entry_init_cmplx_ptr
public :: var_entry_init_pdg_array_ptr
public :: var_entry_init_prt_list_ptr
public :: var_entry_init_string_ptr
```

⟨*Variables: procedures*⟩+≡

```
subroutine var_entry_init_log_ptr (var, name, lval, is_known, intrinsic)
  type(var_entry_t), intent(out) :: var
  type(string_t), intent(in) :: name
  logical, intent(in), target :: lval
  logical, intent(in), target :: is_known
  logical, intent(in), optional :: intrinsic
  var%name = name
  var%type = V_LOG
  var%lval => lval
  var%is_known => is_known
  if (present (intrinsic))  var%is_intrinsic = intrinsic
end subroutine var_entry_init_log_ptr

subroutine var_entry_init_int_ptr (var, name, ival, is_known, intrinsic)
  type(var_entry_t), intent(out) :: var
  type(string_t), intent(in) :: name
  integer, intent(in), target :: ival
  logical, intent(in), target :: is_known
  logical, intent(in), optional :: intrinsic
  var%name = name
  var%type = V_INT
  var%ival => ival
  var%is_known => is_known
```

```
      if (present (intrinsic))  var%is_intrinsic = intrinsic
    end subroutine var_entry_init_int_ptr

    subroutine var_entry_init_real_ptr (var, name, rval, is_known, intrinsic)
      type(var_entry_t), intent(out) :: var
      type(string_t), intent(in) :: name
      real(default), intent(in), target :: rval
      logical, intent(in), target :: is_known
      logical, intent(in), optional :: intrinsic
      var%name = name
      var%type = V_REAL
      var%rval => rval
      var%is_known => is_known
      if (present (intrinsic))  var%is_intrinsic = intrinsic
    end subroutine var_entry_init_real_ptr

    subroutine var_entry_init_cmplx_ptr (var, name, cval, is_known, intrinsic)
      type(var_entry_t), intent(out) :: var
      type(string_t), intent(in) :: name
      complex(default), intent(in), target :: cval
      logical, intent(in), target :: is_known
      logical, intent(in), optional :: intrinsic
      var%name = name
      var%type = V_CMPLX
      var%cval => cval
      var%is_known => is_known
      if (present (intrinsic))  var%is_intrinsic = intrinsic
    end subroutine var_entry_init_cmplx_ptr

    subroutine var_entry_init_pdg_array_ptr (var, name, aval, is_known, intrinsic)
      type(var_entry_t), intent(out) :: var
      type(string_t), intent(in) :: name
      type(pdg_array_t), intent(in), target :: aval
      logical, intent(in), target :: is_known
      logical, intent(in), optional :: intrinsic
      var%name = name
      var%type = V_PDG
      var%aval => aval
      var%is_known => is_known
      if (present (intrinsic))  var%is_intrinsic = intrinsic
    end subroutine var_entry_init_pdg_array_ptr

    subroutine var_entry_init_prt_list_ptr (var, name, pval, is_known, intrinsic)
      type(var_entry_t), intent(out) :: var
      type(string_t), intent(in) :: name
      type(prt_list_t), intent(in), target :: pval
      logical, intent(in), target :: is_known
      logical, intent(in), optional :: intrinsic
      var%name = name
      var%type = V_PTL
      var%pval => pval
      var%is_known => is_known
      if (present (intrinsic))  var%is_intrinsic = intrinsic
    end subroutine var_entry_init_prt_list_ptr
```

```
subroutine var_entry_init_string_ptr (var, name, sval, is_known, intrinsic)
  type(var_entry_t), intent(out) :: var
  type(string_t), intent(in) :: name
  type(string_t), intent(in), target :: sval
  logical, intent(in), target :: is_known
  logical, intent(in), optional :: intrinsic
  var%name = name
  var%type = V_STR
  var%sval => sval
  var%is_known => is_known
  if (present (intrinsic))  var%is_intrinsic = intrinsic
end subroutine var_entry_init_string_ptr
```

Initialize an entry with an observable. The procedure pointer is not yet set.

⟨Variables: procedures⟩+≡
```
subroutine var_entry_init_obs (var, name, type, prt1, prt2)
  type(var_entry_t), intent(out) :: var
  type(string_t), intent(in) :: name
  integer, intent(in) :: type
  type(prt_t), intent(in), target :: prt1
  type(prt_t), intent(in), optional, target :: prt2
  var%type = type
  var%name = name
  var%prt1 => prt1
  if (present (prt2))  var%prt2 => prt2
end subroutine var_entry_init_obs
```

Lock an entry: forbid resetting the entry after initialization.

⟨Variables: procedures⟩+≡
```
subroutine var_entry_lock (var, locked)
  type(var_entry_t), intent(inout) :: var
  logical, intent(in), optional :: locked
  if (present (locked)) then
     var%is_locked = locked
  else
     var%is_locked = .true.
  end if
end subroutine var_entry_lock
```

Finalizer

⟨Variables: public⟩+≡
```
public :: var_entry_final
```

⟨Variables: procedures⟩+≡
```
subroutine var_entry_final (var)
  type(var_entry_t), intent(inout) :: var
  if (var%is_allocated) then
     select case (var%type)
     case (V_LOG); deallocate (var%lval)
     case (V_INT); deallocate (var%ival)
```

```
          case (V_REAL);deallocate (var%rval)
          case (V_CMPLX);deallocate (var%cval)
          case (V_PTL); deallocate (var%pval)
          case (V_PDG); deallocate (var%aval)
          case (V_STR); deallocate (var%sval)
          end select
          deallocate (var%is_known)
          var%is_allocated = .false.
       end if
    end subroutine var_entry_final
```

Output

⟨*Variables: public*⟩+≡
```
    public :: var_entry_write
```

⟨*Variables: procedures*⟩+≡
```
    recursive subroutine var_entry_write (var, unit, model_name, show_ptr, &
        intrinsic)
      type(var_entry_t), intent(in) :: var
      integer, intent(in), optional :: unit
      type(string_t), intent(in), optional :: model_name
      logical, intent(in), optional :: show_ptr
      logical, intent(in), optional :: intrinsic
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      if (present (intrinsic)) then
         if (var%is_intrinsic .neqv. intrinsic)  return
      end if
      if (.not. var%is_intrinsic) then
         write (u, "(A,1x)", advance="no")  "[user variable]"
      end if
      if (associated (var%original)) then
         if (present (model_name)) then
            write (u, "(A,A)", advance="no")  char(model_name), "."
         end if
      end if
      write (u, "(A)", advance="no")  char (var%name)
      if (var%is_locked)  write (u, "(A)", advance="no")  "*"
      if (var%is_allocated) then
         write (u, "(A)", advance="no")  " = "
      else if (var%type /= V_NONE) then
         write (u, "(A)", advance="no")  " => "
      end if
      select case (var%type)
      case (V_NONE); write (u, *)
      case (V_LOG)
         if (var%is_known) then
            if (var%lval) then
               write (u, "(A)")  "true"
            else
               write (u, "(A)")  "false"
            end if
```

```
            else
               write (u, "(A)")  "[unknown logical]"
            end if
         case (V_INT)
            if (var%is_known) then
               write (u, *)  var%ival
            else
               write (u, "(A)")  "[unknown integer]"
            end if
         case (V_REAL)
            if (var%is_known) then
               write (u, *)  var%rval
            else
               write (u, "(A)")  "[unknown real]"
            end if
         case (V_CMPLX)
            if (var%is_known) then
               write (u, *)  var%cval
            else
               write (u, "(A)")  "[unknown complex]"
            end if
         case (V_PTL)
            if (var%is_known) then
               call prt_list_write (var%pval, unit, prefix="        ")
            else
               write (u, "(A)")  "[unknown particle list]"
            end if
         case (V_PDG)
            if (var%is_known) then
               call pdg_array_write (var%aval, u);  write (u, *)
            else
               write (u, "(A)")  "[unknown PDG array]"
            end if
         case (V_STR)
            if (var%is_known) then
               write (u, "(A)")  '"' // char (var%sval) // '"'
            else
               write (u, "(A)")  "[unknown string]"
            end if
         case (V_OBS1_INT);  write (u, *) "[int] = unary observable"
         case (V_OBS2_INT);  write (u, *) "[int] = binary observable"
         case (V_OBS1_REAL); write (u, *) "[real] = unary observable"
         case (V_OBS2_REAL); write (u, *) "[real] = binary observable"
         end select
         if (present (show_ptr)) then
            if (show_ptr .and. var%is_copy .and. associated (var%original)) then
               write (u, "(' => ')", advance="no")
               call var_entry_write (var%original, unit)
            end if
         end if
      end subroutine var_entry_write
```

Accessing contents

⟨*Variables: public*⟩+≡
```
    public :: var_entry_get_name
    public :: var_entry_get_type
```

⟨*Variables: procedures*⟩+≡
```
    function var_entry_get_name (var) result (name)
      type(string_t) :: name
      type(var_entry_t), intent(in) :: var
      name = var%name
    end function var_entry_get_name

    function var_entry_get_type (var) result (type)
      integer :: type
      type(var_entry_t), intent(in) :: var
      type = var%type
    end function var_entry_get_type
```

Return true if the variable is locked

⟨*Variables: public*⟩+≡
```
    public :: var_entry_is_locked
```

⟨*Variables: procedures*⟩+≡
```
    function var_entry_is_locked (var) result (locked)
      logical :: locked
      type(var_entry_t), intent(in) :: var
      locked = var%is_locked
    end function var_entry_is_locked
```

Return true if the variable is intrinsic

⟨*Variables: public*⟩+≡
```
    public :: var_entry_is_intrinsic
```

⟨*Variables: procedures*⟩+≡
```
    function var_entry_is_intrinsic (var) result (flag)
      logical :: flag
      type(var_entry_t), intent(in) :: var
      flag = var%is_intrinsic
    end function var_entry_is_intrinsic
```

Return true if the variable is a copy

⟨*Variables: public*⟩+≡
```
    public :: var_entry_is_copy
```

⟨*Variables: procedures*⟩+≡
```
    function var_entry_is_copy (var) result (flag)
      logical :: flag
      type(var_entry_t), intent(in) :: var
      flag = var%is_copy
    end function var_entry_is_copy
```

Return components

⟨*Variables: public*⟩+≡
```
    public :: var_entry_is_known
    public :: var_entry_get_lval
    public :: var_entry_get_ival
    public :: var_entry_get_rval
    public :: var_entry_get_cval
    public :: var_entry_get_aval
    public :: var_entry_get_pval
    public :: var_entry_get_sval
```
⟨*Variables: procedures*⟩+≡
```
    function var_entry_is_known (var) result (flag)
      logical :: flag
      type(var_entry_t), intent(in) :: var
      flag = var%is_known
    end function var_entry_is_known

    function var_entry_get_lval (var) result (lval)
      logical :: lval
      type(var_entry_t), intent(in) :: var
      lval = var%lval
    end function var_entry_get_lval

    function var_entry_get_ival (var) result (ival)
      integer :: ival
      type(var_entry_t), intent(in) :: var
      ival = var%ival
    end function var_entry_get_ival

    function var_entry_get_rval (var) result (rval)
      real(default) :: rval
      type(var_entry_t), intent(in) :: var
      rval = var%rval
    end function var_entry_get_rval

    function var_entry_get_cval (var) result (cval)
      complex(default) :: cval
      type(var_entry_t), intent(in) :: var
      cval = var%cval
    end function var_entry_get_cval

    function var_entry_get_aval (var) result (aval)
      type(pdg_array_t) :: aval
      type(var_entry_t), intent(in) :: var
      aval = var%aval
    end function var_entry_get_aval

    function var_entry_get_pval (var) result (pval)
      type(prt_list_t) :: pval
      type(var_entry_t), intent(in) :: var
      pval = var%pval
    end function var_entry_get_pval

    function var_entry_get_sval (var) result (sval)
```

```
    type(string_t) :: sval
    type(var_entry_t), intent(in) :: var
    sval = var%sval
  end function var_entry_get_sval
```

Return pointers to components

⟨*Variables: public*⟩+≡
```
    public :: var_entry_get_known_ptr
    public :: var_entry_get_lval_ptr
    public :: var_entry_get_ival_ptr
    public :: var_entry_get_rval_ptr
    public :: var_entry_get_cval_ptr
    public :: var_entry_get_aval_ptr
    public :: var_entry_get_pval_ptr
    public :: var_entry_get_sval_ptr
```

⟨*Variables: procedures*⟩+≡
```
  function var_entry_get_known_ptr (var) result (ptr)
    logical, pointer :: ptr
    type(var_entry_t), intent(in), target :: var
    ptr => var%is_known
  end function var_entry_get_known_ptr

  function var_entry_get_lval_ptr (var) result (ptr)
    logical, pointer :: ptr
    type(var_entry_t), intent(in), target :: var
    ptr => var%lval
  end function var_entry_get_lval_ptr

  function var_entry_get_ival_ptr (var) result (ptr)
    integer, pointer :: ptr
    type(var_entry_t), intent(in), target :: var
    ptr => var%ival
  end function var_entry_get_ival_ptr

  function var_entry_get_rval_ptr (var) result (ptr)
    real(default), pointer :: ptr
    type(var_entry_t), intent(in), target :: var
    ptr => var%rval
  end function var_entry_get_rval_ptr

  function var_entry_get_cval_ptr (var) result (ptr)
    complex(default), pointer :: ptr
    type(var_entry_t), intent(in), target :: var
    ptr => var%cval
  end function var_entry_get_cval_ptr

  function var_entry_get_pval_ptr (var) result (ptr)
    type(prt_list_t), pointer :: ptr
    type(var_entry_t), intent(in), target :: var
    ptr => var%pval
  end function var_entry_get_pval_ptr

  function var_entry_get_aval_ptr (var) result (ptr)
```

```
      type(pdg_array_t), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%aval
    end function var_entry_get_aval_ptr

    function var_entry_get_sval_ptr (var) result (ptr)
      type(string_t), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%sval
    end function var_entry_get_sval_ptr
```

⟨*Variables: public*⟩+≡
```
    public :: var_entry_get_prt1_ptr
    public :: var_entry_get_prt2_ptr
```

⟨*Variables: procedures*⟩+≡
```
    function var_entry_get_prt1_ptr (var) result (ptr)
      type(prt_t), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%prt1
    end function var_entry_get_prt1_ptr

    function var_entry_get_prt2_ptr (var) result (ptr)
      type(prt_t), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%prt2
    end function var_entry_get_prt2_ptr
```

We would like to also use functions here (for consistency), but a nagfor bug
temporarily forces use to use subroutines.

⟨*Variables: public*⟩+≡
```
    public :: var_entry_assign_obs1_int_ptr
    public :: var_entry_assign_obs1_real_ptr
    public :: var_entry_assign_obs2_int_ptr
    public :: var_entry_assign_obs2_real_ptr
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_entry_assign_obs1_int_ptr (ptr, var)
      procedure(obs_unary_int), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%obs1_int
    end subroutine var_entry_assign_obs1_int_ptr

    subroutine var_entry_assign_obs1_real_ptr (ptr, var)
      procedure(obs_unary_real), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%obs1_real
    end subroutine var_entry_assign_obs1_real_ptr

    subroutine var_entry_assign_obs2_int_ptr (ptr, var)
      procedure(obs_binary_int), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%obs2_int
    end subroutine var_entry_assign_obs2_int_ptr
```

```
subroutine var_entry_assign_obs2_real_ptr (ptr, var)
  procedure(obs_binary_real), pointer :: ptr
  type(var_entry_t), intent(in), target :: var
  ptr => var%obs2_real
end subroutine var_entry_assign_obs2_real_ptr
```

5.4.2 Setting values

Undefine the value.

⟨*Variables: procedures*⟩+≡
```
subroutine var_entry_undefine (var)
  type(var_entry_t), intent(inout) :: var
  var%is_known = .false.
end subroutine var_entry_undefine
```

⟨*Variables: public*⟩+≡
```
public :: var_entry_set_log
public :: var_entry_set_int
public :: var_entry_set_real
public :: var_entry_set_cmplx
public :: var_entry_set_pdg_array
public :: var_entry_set_prt_list
public :: var_entry_set_string
```

⟨*Variables: procedures*⟩+≡
```
recursive subroutine var_entry_set_log &
     (var, lval, is_known, verbose, model_name)
  type(var_entry_t), intent(inout) :: var
  logical, intent(in) :: lval
  logical, intent(in) :: is_known
  logical, intent(in), optional :: verbose
  type(string_t), intent(in), optional :: model_name
  integer :: u
  u = logfile_unit ()
  var%lval = lval
  var%is_known = is_known
  if (associated (var%original)) then
     call var_entry_set_log (var%original, lval, is_known)
  end if
  if (present (verbose)) then
     if (verbose) then
        call var_entry_write (var, model_name=model_name)
        call var_entry_write (var, model_name=model_name, unit=u)
        flush (u)
     end if
  end if
end subroutine var_entry_set_log

recursive subroutine var_entry_set_int &
     (var, ival, is_known, verbose, model_name)
  type(var_entry_t), intent(inout) :: var
```

```
  integer, intent(in) :: ival
  logical, intent(in) :: is_known
  logical, intent(in), optional :: verbose
  type(string_t), intent(in), optional :: model_name
  integer :: u
  u = logfile_unit ()
  var%ival = ival
  var%is_known = is_known
  if (associated (var%original)) then
     call var_entry_set_int (var%original, ival, is_known)
  end if
  if (present (verbose)) then
     if (verbose) then
        call var_entry_write (var, model_name=model_name)
        call var_entry_write (var, model_name=model_name, unit=u)
        flush (u)
     end if
  end if
end subroutine var_entry_set_int

recursive subroutine var_entry_set_real &
     (var, rval, is_known, verbose, model_name)
  type(var_entry_t), intent(inout) :: var
  real(default), intent(in) :: rval
  logical, intent(in) :: is_known
  logical, intent(in), optional :: verbose
  type(string_t), intent(in), optional :: model_name
  integer :: u
  u = logfile_unit ()
  var%rval = rval
  var%is_known = is_known
  if (associated (var%original)) then
     call var_entry_set_real (var%original, rval, is_known)
  end if
  if (present (verbose)) then
     if (verbose) then
        call var_entry_write (var, model_name=model_name)
        call var_entry_write (var, model_name=model_name, unit=u)
        flush (u)
     end if
  end if
end subroutine var_entry_set_real

recursive subroutine var_entry_set_cmplx &
     (var, cval, is_known, verbose, model_name)
  type(var_entry_t), intent(inout) :: var
  complex(default), intent(in) :: cval
  logical, intent(in) :: is_known
  logical, intent(in), optional :: verbose
  type(string_t), intent(in), optional :: model_name
  integer :: u
  u = logfile_unit ()
  var%cval = cval
  var%is_known = is_known
```

```fortran
    if (associated (var%original)) then
       call var_entry_set_cmplx (var%original, cval, is_known)
    end if
    if (present (verbose)) then
       if (verbose) then
          call var_entry_write (var, model_name=model_name)
          call var_entry_write (var, model_name=model_name, unit=u)
          flush (u)
       end if
    end if
  end subroutine var_entry_set_cmplx

  recursive subroutine var_entry_set_pdg_array &
       (var, aval, is_known, verbose, model_name)
    type(var_entry_t), intent(inout) :: var
    type(pdg_array_t), intent(in) :: aval
    logical, intent(in) :: is_known
    logical, intent(in), optional :: verbose
    type(string_t), intent(in), optional :: model_name
    integer :: u
    u = logfile_unit ()
    var%aval = aval
    var%is_known = is_known
    if (associated (var%original)) then
       call var_entry_set_pdg_array (var%original, aval, is_known)
    end if
    if (present (verbose)) then
       if (verbose) then
          call var_entry_write (var, model_name=model_name)
          call var_entry_write (var, model_name=model_name, unit=u)
          flush (u)
       end if
    end if
  end subroutine var_entry_set_pdg_array

  recursive subroutine var_entry_set_prt_list &
       (var, pval, is_known, verbose, model_name)
    type(var_entry_t), intent(inout) :: var
    type(prt_list_t), intent(in) :: pval
    logical, intent(in) :: is_known
    logical, intent(in), optional :: verbose
    type(string_t), intent(in), optional :: model_name
    integer :: u
    u = logfile_unit ()
    var%pval = pval
    var%is_known = is_known
    if (associated (var%original)) then
       call var_entry_set_prt_list (var%original, pval, is_known)
    end if
    if (present (verbose)) then
       if (verbose) then
          call var_entry_write (var, model_name=model_name)
          call var_entry_write (var, model_name=model_name, unit=u)
          flush (u)
```

```
      end if
   end if
end subroutine var_entry_set_prt_list

recursive subroutine var_entry_set_string &
     (var, sval, is_known, verbose, model_name)
  type(var_entry_t), intent(inout) :: var
  type(string_t), intent(in) :: sval
  logical, intent(in) :: is_known
  logical, intent(in), optional :: verbose
  type(string_t), intent(in), optional :: model_name
  integer :: u
  u = logfile_unit ()
  var%sval = sval
  var%is_known = is_known
  if (associated (var%original)) then
     call var_entry_set_string (var%original, sval, is_known)
  end if
  if (present (verbose)) then
     if (verbose) then
        call var_entry_write (var, model_name=model_name)
        call var_entry_write (var, model_name=model_name, unit=u)
        flush (u)
     end if
  end if
end subroutine var_entry_set_string
```

5.4.3   Copies and pointer variables

Initialize an entry with a copy of an existing variable entry. The copy is physically allocated with the same type as the original.

⟨*Variables: procedures*⟩+≡

```
    subroutine var_entry_init_copy (var, original)
      type(var_entry_t), intent(out) :: var
      type(var_entry_t), intent(in), target :: original
      type(string_t) :: name
      logical :: intrinsic
      name = var_entry_get_name (original)
      intrinsic = original%is_intrinsic
      select case (original%type)
      case (V_LOG)
         call var_entry_init_log (var, name, intrinsic=intrinsic)
      case (V_INT)
         call var_entry_init_int (var, name, intrinsic=intrinsic)
      case (V_REAL)
         call var_entry_init_real (var, name, intrinsic=intrinsic)
      case (V_CMPLX)
         call var_entry_init_cmplx (var, name, intrinsic=intrinsic)
      case (V_PTL)
         call var_entry_init_prt_list (var, name, intrinsic=intrinsic)
      case (V_PDG)
         call var_entry_init_pdg_array (var, name, intrinsic=intrinsic)
```

```
      case (V_STR)
         call var_entry_init_string (var, name, intrinsic=intrinsic)
      end select
      var%is_copy = .true.
    end subroutine var_entry_init_copy
```

Clear the pointer to the original.

⟨*Variables: procedures*⟩+≡
```
    subroutine var_entry_clear_original_pointer (var)
      type(var_entry_t), intent(inout) :: var
      var%original => null ()
    end subroutine var_entry_clear_original_pointer
```

Set the pointer to the original. For a free parameter, the variable holds both the value and the pointer. For a derived parameter, we associate the pointer directly. Derived parameters thus need not be synchronized explicitly.

Update: this does not work. If locked parameters are accessed in expressions and the model is non-default, the pointers in the expression may be undefined at compile time. Reassigning the variables at runtime does not help, since the pointers in the expression are dereferenced before assignment. Hence, no special treatment for derived parameters.

⟨*Variables: procedures*⟩+≡
```
    subroutine var_entry_set_original_pointer (var, original)
      type(var_entry_t), intent(inout) :: var
      type(var_entry_t), intent(in), target :: original
      type(string_t) :: name
      type(var_entry_t), pointer :: next
      if (var_entry_is_locked (original)) then
!        next => var%next
!        name = var_entry_get_name (original)
!        select case (original%type)
!        case (V_LOG);   call var_entry_init_log_ptr (var, name, &
!             original%lval, original%is_known)
!        case (V_INT);   call var_entry_init_int_ptr (var, name, &
!             original%ival, original%is_known)
!        case (V_REAL);  call var_entry_init_real_ptr (var, name, &
!             original%rval, original%is_known)
!        case (V_CMPLX); call var_entry_init_cmplx_ptr (var, name, &
!             original%cval, original%is_known)
!        case (V_PTL);   call var_entry_init_prt_list_ptr (var, name, &
!             original%pval, original%is_known)
!        case (V_PDG);   call var_entry_init_pdg_array_ptr (var, name, &
!             original%aval, original%is_known)
!        case (V_STR);   call var_entry_init_string_ptr (var, name, &
!             original%sval, original%is_known)
!        end select
!        var%next => next
         call var_entry_lock (var)
!      else
!         var%original => original
      end if
      var%original => original
```

```
      end subroutine var_entry_set_original_pointer
```

Synchronize a variable with its original: if a pointer exists, set the value to be equal to value pointed to.

⟨*Variables: procedures*⟩+≡
```
   subroutine var_entry_synchronize (var)
     type(var_entry_t), intent(inout) :: var
     if (associated (var%original)) then
        var%is_known = var%original%is_known
        if (var%original%is_known) then
           select case (var%type)
           case (V_LOG);   var%lval = var%original%lval
           case (V_INT);   var%ival = var%original%ival
           case (V_REAL); var%rval = var%original%rval
           case (V_CMPLX); var%cval = var%original%cval
           case (V_PTL);   var%pval = var%original%pval
           case (V_PDG);   var%aval = var%original%aval
           case (V_STR);   var%sval = var%original%sval
           end select
        end if
     end if
   end subroutine var_entry_synchronize
```

Restore the previous value of the original, using the value stored in the variable. This is a side-effect operation.

⟨*Variables: procedures*⟩+≡
```
   subroutine var_entry_restore (var)
     type(var_entry_t), intent(inout) :: var
     !!! ifort 11.1 rev5 chokes over the intent(in)
     !!! type(var_entry_t), intent(in) :: var
     if (associated (var%original)) then
        if (var%is_known) then
           select case (var%type)
           case (V_LOG);   var%original%lval = var%lval
           case (V_INT);   var%original%ival = var%ival
           case (V_REAL); var%original%rval = var%rval
           case (V_CMPLX); var%original%cval = var%cval
           case (V_PTL);   var%original%pval = var%pval
           case (V_PDG);   var%original%aval = var%aval
           case (V_STR);   var%original%sval = var%sval
           end select
        end if
     end if
   end subroutine var_entry_restore
```

5.4.4   Variable lists

The type

Variable lists can be linked together. No initializer needed. They are deleted separately.

⟨*Variables: public*⟩+≡

```
      public :: var_list_t
```

⟨Variables: types⟩+≡
```
    type :: var_list_t
       private
       type(var_entry_t), pointer :: first => null ()
       type(var_entry_t), pointer :: last => null ()
       type(var_list_t), pointer :: next => null ()
    end type var_list_t
```

Constructors

⟨Variables: public⟩+≡
```
    public :: var_list_link
```

⟨Variables: procedures⟩+≡
```
    subroutine var_list_link (var_list, next)
      type(var_list_t), intent(inout) :: var_list
      type(var_list_t), intent(in), target :: next
      var_list%next => next
    end subroutine var_list_link
```

Append a new entry to an existing list.

⟨Variables: procedures⟩+≡
```
    subroutine var_list_append (var_list, var, verbose)
      type(var_list_t), intent(inout) :: var_list
      type(var_entry_t), intent(in), target :: var
      logical, intent(in), optional :: verbose
      if (associated (var_list%last)) then
         var_list%last%next => var
      else
         var_list%first => var
      end if
      var_list%last => var
      if (present (verbose)) then
         if (verbose)  call var_entry_write (var)
      end if
    end subroutine var_list_append
```

⟨Variables: public⟩+≡
```
    public :: var_list_append_log
    public :: var_list_append_int
    public :: var_list_append_real
    public :: var_list_append_cmplx
    public :: var_list_append_prt_list
    public :: var_list_append_pdg_array
    public :: var_list_append_string
```

⟨Variables: procedures⟩+≡
```
    subroutine var_list_append_log &
         (var_list, name, lval, locked, verbose, intrinsic)
      type(var_list_t), intent(inout) :: var_list
      type(string_t), intent(in) :: name
```

```
  logical, intent(in), optional :: lval
  logical, intent(in), optional :: locked, verbose, intrinsic
  type(var_entry_t), pointer :: var
  allocate (var)
  call var_entry_init_log (var, name, lval, intrinsic)
  if (present (locked))  call var_entry_lock (var, locked)
  call var_list_append (var_list, var, verbose)
end subroutine var_list_append_log

subroutine var_list_append_int &
     (var_list, name, ival, locked, verbose, intrinsic)
  type(var_list_t), intent(inout) :: var_list
  type(string_t), intent(in) :: name
  integer, intent(in), optional :: ival
  logical, intent(in), optional :: locked, verbose, intrinsic
  type(var_entry_t), pointer :: var
  allocate (var)
  call var_entry_init_int (var, name, ival, intrinsic)
  if (present (locked))  call var_entry_lock (var, locked)
  call var_list_append (var_list, var, verbose)
end subroutine var_list_append_int

subroutine var_list_append_real &
     (var_list, name, rval, locked, verbose, intrinsic)
  type(var_list_t), intent(inout) :: var_list
  type(string_t), intent(in) :: name
  real(default), intent(in), optional :: rval
  logical, intent(in), optional :: locked, verbose, intrinsic
  type(var_entry_t), pointer :: var
  allocate (var)
  call var_entry_init_real (var, name, rval, intrinsic)
  if (present (locked))  call var_entry_lock (var, locked)
  call var_list_append (var_list, var, verbose)
end subroutine var_list_append_real

subroutine var_list_append_cmplx &
     (var_list, name, cval, locked, verbose, intrinsic)
  type(var_list_t), intent(inout) :: var_list
  type(string_t), intent(in) :: name
  complex(default), intent(in), optional :: cval
  logical, intent(in), optional :: locked, verbose, intrinsic
  type(var_entry_t), pointer :: var
  allocate (var)
  call var_entry_init_cmplx (var, name, cval, intrinsic)
  if (present (locked))  call var_entry_lock (var, locked)
  call var_list_append (var_list, var, verbose)
end subroutine var_list_append_cmplx

subroutine var_list_append_prt_list &
     (var_list, name, pval, locked, verbose, intrinsic)
  type(var_list_t), intent(inout) :: var_list
  type(string_t), intent(in) :: name
  type(prt_list_t), intent(in), optional :: pval
  logical, intent(in), optional :: locked, verbose, intrinsic
```

```
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_prt_list (var, name, pval, intrinsic)
    if (present (locked))  call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
  end subroutine var_list_append_prt_list

  subroutine var_list_append_pdg_array &
       (var_list, name, aval, locked, verbose, intrinsic)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    type(pdg_array_t), intent(in), optional :: aval
    logical, intent(in), optional :: locked, verbose, intrinsic
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_pdg_array (var, name, aval, intrinsic)
    if (present (locked))  call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
  end subroutine var_list_append_pdg_array

  subroutine var_list_append_string &
       (var_list, name, sval, locked, verbose, intrinsic)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    type(string_t), intent(in), optional :: sval
    logical, intent(in), optional :: locked, verbose, intrinsic
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_string (var, name, sval, intrinsic)
    if (present (locked))  call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
  end subroutine var_list_append_string
```

⟨*Variables: public*⟩+≡
```
    public :: var_list_append_log_ptr
    public :: var_list_append_int_ptr
    public :: var_list_append_real_ptr
    public :: var_list_append_cmplx_ptr
    public :: var_list_append_pdg_array_ptr
    public :: var_list_append_prt_list_ptr
    public :: var_list_append_string_ptr
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_append_log_ptr &
         (var_list, name, lval, is_known, locked, verbose, intrinsic)
      type(var_list_t), intent(inout) :: var_list
      type(string_t), intent(in) :: name
      logical, intent(in), target :: lval
      logical, intent(in), target :: is_known
      logical, intent(in), optional :: locked, verbose, intrinsic
      type(var_entry_t), pointer :: var
      allocate (var)
      call var_entry_init_log_ptr (var, name, lval, is_known, intrinsic)
      if (present (locked))  call var_entry_lock (var, locked)
```

```
    call var_list_append (var_list, var, verbose)
end subroutine var_list_append_log_ptr

subroutine var_list_append_int_ptr &
     (var_list, name, ival, is_known, locked, verbose, intrinsic)
  type(var_list_t), intent(inout) :: var_list
  type(string_t), intent(in) :: name
  integer, intent(in), target :: ival
  logical, intent(in), target :: is_known
  logical, intent(in), optional :: locked, verbose, intrinsic
  type(var_entry_t), pointer :: var
  allocate (var)
  call var_entry_init_int_ptr (var, name, ival, is_known, intrinsic)
  if (present (locked))  call var_entry_lock (var, locked)
  call var_list_append (var_list, var, verbose)
end subroutine var_list_append_int_ptr

subroutine var_list_append_real_ptr &
     (var_list, name, rval, is_known, locked, verbose, intrinsic)
  type(var_list_t), intent(inout) :: var_list
  type(string_t), intent(in) :: name
  real(default), intent(in), target :: rval
  logical, intent(in), target :: is_known
  logical, intent(in), optional :: locked, verbose, intrinsic
  type(var_entry_t), pointer :: var
  allocate (var)
  call var_entry_init_real_ptr (var, name, rval, is_known, intrinsic)
  if (present (locked))  call var_entry_lock (var, locked)
  call var_list_append (var_list, var, verbose)
end subroutine var_list_append_real_ptr

subroutine var_list_append_cmplx_ptr &
     (var_list, name, cval, is_known, locked, verbose, intrinsic)
  type(var_list_t), intent(inout) :: var_list
  type(string_t), intent(in) :: name
  complex(default), intent(in), target :: cval
  logical, intent(in), target :: is_known
  logical, intent(in), optional :: locked, verbose, intrinsic
  type(var_entry_t), pointer :: var
  allocate (var)
  call var_entry_init_cmplx_ptr (var, name, cval, is_known, intrinsic)
  if (present (locked))  call var_entry_lock (var, locked)
  call var_list_append (var_list, var, verbose)
end subroutine var_list_append_cmplx_ptr

subroutine var_list_append_pdg_array_ptr &
     (var_list, name, aval, is_known, locked, verbose, intrinsic)
  type(var_list_t), intent(inout) :: var_list
  type(string_t), intent(in) :: name
  type(pdg_array_t), intent(in), target :: aval
  logical, intent(in), target :: is_known
  logical, intent(in), optional :: locked, verbose, intrinsic
  type(var_entry_t), pointer :: var
  allocate (var)
```

```
      call var_entry_init_pdg_array_ptr (var, name, aval, is_known, intrinsic)
      if (present (locked))  call var_entry_lock (var, locked)
      call var_list_append (var_list, var, verbose)
    end subroutine var_list_append_pdg_array_ptr

    subroutine var_list_append_prt_list_ptr &
         (var_list, name, pval, is_known, locked, verbose, intrinsic)
      type(var_list_t), intent(inout) :: var_list
      type(string_t), intent(in) :: name
      type(prt_list_t), intent(in), target :: pval
      logical, intent(in), target :: is_known
      logical, intent(in), optional :: locked, verbose, intrinsic
      type(var_entry_t), pointer :: var
      allocate (var)
      call var_entry_init_prt_list_ptr (var, name, pval, is_known, intrinsic)
      if (present (locked))  call var_entry_lock (var, locked)
      call var_list_append (var_list, var, verbose)
    end subroutine var_list_append_prt_list_ptr

    subroutine var_list_append_string_ptr &
         (var_list, name, sval, is_known, locked, verbose, intrinsic)
      type(var_list_t), intent(inout) :: var_list
      type(string_t), intent(in) :: name
      type(string_t), intent(in), target :: sval
      logical, intent(in), target :: is_known
      logical, intent(in), optional :: locked, verbose, intrinsic
      type(var_entry_t), pointer :: var
      allocate (var)
      call var_entry_init_string_ptr (var, name, sval, is_known, intrinsic)
      if (present (locked))  call var_entry_lock (var, locked)
      call var_list_append (var_list, var, verbose)
    end subroutine var_list_append_string_ptr
```

Finalizer

⟨*Variables: public*⟩+≡
```
    public :: var_list_final
```

Finalize, delete the list entry by entry.

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_final (var_list)
      type(var_list_t), intent(inout) :: var_list
      type(var_entry_t), pointer :: var
      var_list%last => null ()
      do while (associated (var_list%first))
         var => var_list%first
         var_list%first => var%next
         call var_entry_final (var)
         deallocate (var)
      end do
    end subroutine var_list_final
```

Output

Optionally, show only variables of a certain type.

⟨*Variables: public*⟩+≡
```
    public :: var_list_write
```

⟨*Variables: procedures*⟩+≡
```
    recursive subroutine var_list_write &
        (var_list, unit, follow_link, only_type, prefix, model_name, show_ptr, &
         intrinsic)
      type(var_list_t), intent(in), target :: var_list
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: follow_link
      integer, intent(in), optional :: only_type
      character(*), intent(in), optional :: prefix
      type(string_t), intent(in), optional :: model_name
      logical, intent(in), optional :: show_ptr
      logical, intent(in), optional :: intrinsic
      type(var_entry_t), pointer :: var
      integer :: u, length
      logical :: write_this, write_next
      u = output_unit (unit);  if (u < 0)  return
      if (present (prefix))  length = len (prefix)
      var => var_list%first
      if (associated (var)) then
         do while (associated (var))
            if (present (only_type)) then
               write_this = only_type == var%type
            else
               write_this = .true.
            end if
            if (write_this .and. present (prefix)) then
               if (prefix /= extract (var%name, 1, length)) &
                    write_this = .false.
            end if
            if (write_this) then
               call var_entry_write &
                    (var, unit, model_name = model_name, show_ptr = show_ptr, &
                     intrinsic=intrinsic)
            end if
            var => var%next
         end do
      end if
      write_next = associated (var_list%next)
      if (present (follow_link)) &
           write_next = write_next .and. follow_link
      if (write_next) then
         call var_list_write (var_list%next, &
              unit, follow_link, only_type, prefix, model_name, show_ptr, &
              intrinsic)
      end if
    end subroutine var_list_write
```

Write only a certain variable.

⟨*Variables: public*⟩+≡
```
    public :: var_list_write_var
```

⟨*Variables: procedures*⟩+≡
```
    recursive subroutine var_list_write_var &
        (var_list, name, unit, type, follow_link, model_name, show_ptr)
      type(var_list_t), intent(in), target :: var_list
      type(string_t), intent(in) :: name
      integer, intent(in), optional :: unit
      integer, intent(in), optional :: type
      logical, intent(in), optional :: follow_link
      type(string_t), intent(in), optional :: model_name
      logical, intent(in), optional :: show_ptr
      type(var_entry_t), pointer :: var
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      var => var_list_get_var_ptr (var_list, name, type, follow_link=follow_link)
      if (associated (var)) then
         call var_entry_write &
              (var, unit, model_name = model_name, show_ptr = show_ptr)
      else
         write (u, "(A)")  char (name) // " = [undefined]"
      end if
    end subroutine var_list_write_var
```

5.4.5 Tools

Return a pointer to the variable with the requested name. If no such name exists, return a null pointer. In that case, try the next list if present.

⟨*Variables: public*⟩+≡
```
    public :: var_list_get_var_ptr
```

⟨*Variables: procedures*⟩+≡
```
    recursive function var_list_get_var_ptr (var_list, name, type, follow_link) &
        result (var)
      type(var_entry_t), pointer :: var
      type(var_list_t), intent(in), target :: var_list
      type(string_t), intent(in) :: name
      integer, intent(in), optional :: type
      logical, intent(in), optional :: follow_link
      logical :: search_next
      var => var_list%first
      if (present (type)) then
         do while (associated (var))
            if (var%type == type) then
               if (var%name == name)  return
            end if
            var => var%next
         end do
      else
         do while (associated (var))
            if (var%name == name)  return
            var => var%next
```

```
            end do
        end if
        search_next = associated (var_list%next)
        if (present (follow_link)) &
            search_next = search_next .and. follow_link
        if (search_next) &
            var => var_list_get_var_ptr (var_list%next, name, type)
    end function var_list_get_var_ptr
```

Return the variable type

⟨*Variables: public*⟩+≡
```
    public :: var_list_get_type
```

⟨*Variables: procedures*⟩+≡
```
    function var_list_get_type (var_list, name, follow_link) result (type)
      integer :: type
      type(string_t), intent(in) :: name
      type(var_list_t), intent(in), target :: var_list
      logical, intent(in), optional :: follow_link
      type(var_entry_t), pointer :: var
      var => var_list_get_var_ptr (var_list, name, follow_link=follow_link)
      if (associated (var)) then
          type = var%type
      else
          type = V_NONE
      end if
    end function var_list_get_type
```

Return true if the variable exists.

⟨*Variables: public*⟩+≡
```
    public :: var_list_exists
```

⟨*Variables: procedures*⟩+≡
```
    function var_list_exists (var_list, name, follow_link) result (flag)
      logical :: flag
      type(string_t), intent(in) :: name
      type(var_list_t), intent(in), target :: var_list
      logical, intent(in), optional :: follow_link
      type(var_entry_t), pointer :: var
      var => var_list_get_var_ptr (var_list, name, follow_link=follow_link)
      flag = associated (var)
    end function var_list_exists
```

Return true if the variable is declared as intrinsic.

⟨*Variables: public*⟩+≡
```
    public :: var_list_is_intrinsic
```

⟨*Variables: procedures*⟩+≡
```
    function var_list_is_intrinsic (var_list, name, follow_link) result (flag)
      logical :: flag
      type(string_t), intent(in) :: name
      type(var_list_t), intent(in), target :: var_list
      logical, intent(in), optional :: follow_link
      type(var_entry_t), pointer :: var
```

```
        var => var_list_get_var_ptr (var_list, name, follow_link=follow_link)
        if (associated (var)) then
           flag = var%is_intrinsic
        else
           flag = .false.
        end if
     end function var_list_is_intrinsic
```

Return true if the value is known.

⟨*Variables: public*⟩+≡
```
     public :: var_list_is_known
```

⟨*Variables: procedures*⟩+≡
```
     function var_list_is_known (var_list, name, follow_link) result (flag)
       logical :: flag
       type(string_t), intent(in) :: name
       type(var_list_t), intent(in), target :: var_list
       logical, intent(in), optional :: follow_link
       type(var_entry_t), pointer :: var
       var => var_list_get_var_ptr (var_list, name, follow_link=follow_link)
       if (associated (var)) then
          flag = var%is_known
       else
          flag = .false.
       end if
     end function var_list_is_known
```

Return true if the value is locked.

⟨*Variables: public*⟩+≡
```
     public :: var_list_is_locked
```

⟨*Variables: procedures*⟩+≡
```
     function var_list_is_locked (var_list, name, follow_link) result (flag)
       logical :: flag
       type(string_t), intent(in) :: name
       type(var_list_t), intent(in), target :: var_list
       logical, intent(in), optional :: follow_link
       type(var_entry_t), pointer :: var
       var => var_list_get_var_ptr (var_list, name, follow_link=follow_link)
       if (associated (var)) then
          flag = var_entry_is_locked (var)
       else
          flag = .false.
       end if
     end function var_list_is_locked
```

Return the value, assuming that the type is correct:

⟨*Variables: public*⟩+≡
```
     public :: var_list_get_lval
     public :: var_list_get_ival
     public :: var_list_get_rval
     public :: var_list_get_cval
     public :: var_list_get_pval
```

```
      public :: var_list_get_aval
      public :: var_list_get_sval
```
⟨*Variables: procedures*⟩+≡
```
      function var_list_get_lval (var_list, name, follow_link) result (lval)
        logical :: lval
        type(string_t), intent(in) :: name
        type(var_list_t), intent(in), target :: var_list
        logical, intent(in), optional :: follow_link
        type(var_entry_t), pointer :: var
        var => var_list_get_var_ptr (var_list, name, V_LOG, follow_link)
        lval = var%lval
      end function var_list_get_lval

      function var_list_get_ival (var_list, name, follow_link) result (ival)
        integer :: ival
        type(string_t), intent(in) :: name
        type(var_list_t), intent(in), target :: var_list
        logical, intent(in), optional :: follow_link
        type(var_entry_t), pointer :: var
        var => var_list_get_var_ptr (var_list, name, V_INT, follow_link)
        ival = var%ival
      end function var_list_get_ival

      function var_list_get_rval (var_list, name, follow_link) result (rval)
        real(default) :: rval
        type(string_t), intent(in) :: name
        type(var_list_t), intent(in), target :: var_list
        logical, intent(in), optional :: follow_link
        type(var_entry_t), pointer :: var
        var => var_list_get_var_ptr (var_list, name, V_REAL, follow_link)
        rval = var%rval
      end function var_list_get_rval

      function var_list_get_cval (var_list, name, follow_link) result (cval)
        complex(default) :: cval
        type(string_t), intent(in) :: name
        type(var_list_t), intent(in), target :: var_list
        logical, intent(in), optional :: follow_link
        type(var_entry_t), pointer :: var
        var => var_list_get_var_ptr (var_list, name, V_CMPLX, follow_link)
        cval = var%cval
      end function var_list_get_cval

      function var_list_get_aval (var_list, name, follow_link) result (aval)
        type(pdg_array_t) :: aval
        type(string_t), intent(in) :: name
        type(var_list_t), intent(in), target :: var_list
        logical, intent(in), optional :: follow_link
        type(var_entry_t), pointer :: var
        var => var_list_get_var_ptr (var_list, name, V_PDG, follow_link)
        aval = var%aval
      end function var_list_get_aval

      function var_list_get_pval (var_list, name, follow_link) result (pval)
```

```
  type(prt_list_t) :: pval
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in), target :: var_list
  logical, intent(in), optional :: follow_link
  type(var_entry_t), pointer :: var
  var => var_list_get_var_ptr (var_list, name, V_PTL, follow_link)
  pval = var%pval
end function var_list_get_pval

function var_list_get_sval (var_list, name, follow_link) result (sval)
  type(string_t) :: sval
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in), target :: var_list
  logical, intent(in), optional :: follow_link
  type(var_entry_t), pointer :: var
  var => var_list_get_var_ptr (var_list, name, V_STR, follow_link)
  sval = var%sval
end function var_list_get_sval
```

5.4.6 Result variables

Integration results are stored in special variables. They are initialized by this
subroutine. The values may or may not already known.

⟨*Variables: public*⟩+≡
```
    public :: var_list_init_process_results
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_init_process_results (var_list, proc_id, &
        n_calls, integral, error, accuracy, chi2, efficiency)
      type(var_list_t), intent(inout) :: var_list
      type(string_t), intent(in) :: proc_id
      integer, intent(in), optional :: n_calls
      real(default), intent(in), optional :: integral, error, accuracy
      real(default), intent(in), optional :: chi2, efficiency
      call set_var_int (var_str ("n_calls"), n_calls)
      call set_var_real (var_str ("integral"), integral)
      call set_var_real (var_str ("error"), error)
      call set_var_real (var_str ("accuracy"), accuracy)
      call set_var_real (var_str ("chi2"), chi2)
      call set_var_real (var_str ("efficiency"), efficiency)
    contains
      subroutine set_var_int (name, ival)
        type(string_t), intent(in) :: name
        integer, intent(in), optional :: ival
        type(string_t) :: var_name
        type(var_entry_t), pointer :: var
        var_name = name // "(" // proc_id // ")"
        var => var_list_get_var_ptr (var_list, var_name)
        if (.not. associated (var)) then
           call var_list_append_int (var_list, var_name, ival, intrinsic=.true.)
        else if (present (ival)) then
           call var_list_set_int (var_list, var_name, ival, is_known=.true.)
        end if
```

```
      end subroutine set_var_int
    subroutine set_var_real (name, rval)
      type(string_t), intent(in) :: name
      real(default), intent(in), optional :: rval
      type(string_t) :: var_name
      type(var_entry_t), pointer :: var
      var_name = name // "(" // proc_id // ")"
      var => var_list_get_var_ptr (var_list, var_name)
      if (.not. associated (var)) then
         call var_list_append_real (var_list, var_name, rval, intrinsic=.true.)
      else if (present (rval)) then
         call var_list_set_real (var_list, var_name, rval, is_known=.true.)
      end if
    end subroutine set_var_real
  end subroutine var_list_init_process_results
```

5.4.7   Observable initialization

Observables are formally treated as variables, which however are evaluated each
time the observable is used. The arguments (pointers) to evaluate and the
function are part of the variable-list entry.

   The procedure pointer should be set by this subroutine. This, however,
triggers a bug in nagfor 5.2(649). As a workaround, we return the variable
pointer, so the pointer can be set directly.

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_set_obs (var_list, name, type, var, prt1, prt2)
      type(var_list_t), intent(inout) :: var_list
      type(string_t), intent(in) :: name
      integer, intent(in) :: type
      type(var_entry_t), pointer :: var
      type(prt_t), intent(in), target :: prt1
      type(prt_t), intent(in), optional, target :: prt2
      allocate (var)
      call var_entry_init_obs (var, name, type, prt1, prt2)
      call var_list_append (var_list, var)
    end subroutine var_list_set_obs
```

Unary and binary observables are different. Most unary observables can be
equally well evaluated for particle pairs. Binary observables cannot be evaluated
for single particles.

⟨*Variables: public*⟩+≡
```
    public :: var_list_set_observables_unary
    public :: var_list_set_observables_binary
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_set_observables_unary (var_list, prt1)
      type(var_list_t), intent(inout) :: var_list
      type(prt_t), intent(in), target :: prt1
      type(var_entry_t), pointer :: var
      call var_list_set_obs &
           (var_list, var_str ("PDG"), V_OBS1_INT, var, prt1)
      var% obs1_int => obs_pdg1
```

```
      call var_list_set_obs &
           (var_list, var_str ("Hel"), V_OBS1_INT, var, prt1)
      var% obs1_int => obs_helicity1
      call var_list_set_obs &
           (var_list, var_str ("M"), V_OBS1_REAL, var, prt1)
      var% obs1_real => obs_signed_mass1
      call var_list_set_obs &
           (var_list, var_str ("M2"), V_OBS1_REAL, var, prt1)
      var% obs1_real => obs_mass_squared1
      call var_list_set_obs &
           (var_list, var_str ("E"), V_OBS1_REAL, var, prt1)
      var% obs1_real => obs_energy1
      call var_list_set_obs &
           (var_list, var_str ("Px"), V_OBS1_REAL, var, prt1)
      var% obs1_real => obs_px1
      call var_list_set_obs &
           (var_list, var_str ("Py"), V_OBS1_REAL, var, prt1)
      var% obs1_real => obs_py1
      call var_list_set_obs &
           (var_list, var_str ("Pz"), V_OBS1_REAL, var, prt1)
      var% obs1_real => obs_pz1
      call var_list_set_obs &
           (var_list, var_str ("P"), V_OBS1_REAL, var, prt1)
      var% obs1_real => obs_p1
      call var_list_set_obs &
           (var_list, var_str ("Pl"), V_OBS1_REAL, var, prt1)
      var% obs1_real => obs_pl1
      call var_list_set_obs &
           (var_list, var_str ("Pt"), V_OBS1_REAL, var, prt1)
      var% obs1_real => obs_pt1
      call var_list_set_obs &
           (var_list, var_str ("Theta"), V_OBS1_REAL, var, prt1)
      var% obs1_real => obs_theta1
      call var_list_set_obs &
           (var_list, var_str ("Phi"), V_OBS1_REAL, var, prt1)
      var% obs1_real => obs_phi1
      call var_list_set_obs &
           (var_list, var_str ("Rap"), V_OBS1_REAL, var, prt1)
      var% obs1_real => obs_rap1
      call var_list_set_obs &
           (var_list, var_str ("Eta"), V_OBS1_REAL, var, prt1)
      var% obs1_real => obs_eta1
      call var_list_set_obs &
           (var_list, var_str ("Theta_RF"), V_OBS1_REAL, var, prt1)
      var% obs1_real => obs_theta_rf1
      call var_list_set_obs &
           (var_list, var_str ("Dist"), V_OBS1_REAL, var, prt1)
      var% obs1_real => obs_dist1
    end subroutine var_list_set_observables_unary

    subroutine var_list_set_observables_binary (var_list, prt1, prt2)
      type(var_list_t), intent(inout) :: var_list
      type(prt_t), intent(in), target :: prt1
      type(prt_t), intent(in), optional, target :: prt2
```

```
      type(var_entry_t), pointer :: var
      call var_list_set_obs &
           (var_list, var_str ("PDG"), V_OBS2_INT, var, prt1, prt2)
      var% obs2_int => obs_pdg2
      call var_list_set_obs &
           (var_list, var_str ("Hel"), V_OBS2_INT, var, prt1, prt2)
      var% obs2_int => obs_helicity2
      call var_list_set_obs &
           (var_list, var_str ("M"), V_OBS2_REAL, var, prt1, prt2)
      var% obs2_real => obs_signed_mass2
      call var_list_set_obs &
           (var_list, var_str ("M2"), V_OBS2_REAL, var, prt1, prt2)
      var% obs2_real => obs_mass_squared2
      call var_list_set_obs &
           (var_list, var_str ("E"), V_OBS2_REAL, var, prt1, prt2)
      var% obs2_real => obs_energy2
      call var_list_set_obs &
           (var_list, var_str ("Px"), V_OBS2_REAL, var, prt1, prt2)
      var% obs2_real => obs_px2
      call var_list_set_obs &
           (var_list, var_str ("Py"), V_OBS2_REAL, var, prt1, prt2)
      var% obs2_real => obs_py2
      call var_list_set_obs &
           (var_list, var_str ("Pz"), V_OBS2_REAL, var, prt1, prt2)
      var% obs2_real => obs_pz2
      call var_list_set_obs &
           (var_list, var_str ("P"), V_OBS2_REAL, var, prt1, prt2)
      var% obs2_real => obs_p2
      call var_list_set_obs &
           (var_list, var_str ("Pl"), V_OBS2_REAL, var, prt1, prt2)
      var% obs2_real => obs_pl2
      call var_list_set_obs &
           (var_list, var_str ("Pt"), V_OBS2_REAL, var, prt1, prt2)
      var% obs2_real => obs_pt2
      call var_list_set_obs &
           (var_list, var_str ("Theta"), V_OBS2_REAL, var, prt1, prt2)
      var% obs2_real => obs_theta2
      call var_list_set_obs &
           (var_list, var_str ("Phi"), V_OBS2_REAL, var, prt1, prt2)
      var% obs2_real => obs_phi2
      call var_list_set_obs &
           (var_list, var_str ("Rap"), V_OBS2_REAL, var, prt1, prt2)
      var% obs2_real => obs_rap2
      call var_list_set_obs &
           (var_list, var_str ("Eta"), V_OBS2_REAL, var, prt1, prt2)
      var% obs2_real => obs_eta2
      call var_list_set_obs &
           (var_list, var_str ("Theta_RF"), V_OBS2_REAL, var, prt1, prt2)
      var% obs2_real => obs_theta_rf2
      call var_list_set_obs &
           (var_list, var_str ("Dist"), V_OBS2_REAL, var, prt1, prt2)
      var% obs2_real => obs_dist2
    end subroutine var_list_set_observables_binary
```

Check if a variable name is defined as an observable:

⟨*Variables: public*⟩+≡
```
    public :: string_is_observable_id
```

⟨*Variables: procedures*⟩+≡
```
    function string_is_observable_id (string) result (flag)
      logical :: flag
      type(string_t), intent(in) :: string
      select case (char (string))
      case ("PDG", "Hel", "M", "M2", "E", "Px", "Py", "Pz", "P", "Pl", "Pt", &
            "Theta", "Phi", "Rap", "Eta", "Theta_RF", "Dist")
         flag = .true.
      case default
         flag = .false.
      end select
    end function string_is_observable_id
```

5.4.8   Observables

These are analogous to the unary and binary numeric functions listed above. An observable takes the `pval` component(s) of its one or two argument nodes and produces an integer or real value.

Integer-valued unary observables

The PDG code

⟨*Variables: procedures*⟩+≡
```
    integer function obs_pdg1 (prt1) result (pdg)
      type(prt_t), intent(in) :: prt1
      pdg = prt_get_pdg (prt1)
    end function obs_pdg1
```

The helicity. The return value is meaningful only if the particle is polarized, otherwise an invalid value is returned (-9).

⟨*Variables: procedures*⟩+≡
```
    integer function obs_helicity1 (prt1) result (h)
      type(prt_t), intent(in) :: prt1
      if (prt_is_polarized (prt1)) then
         h = prt_get_helicity (prt1)
      else
         h = -9
      end if
    end function obs_helicity1
```

Real-valued unary observables

The invariant mass squared, obtained from the separately stored value.

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_mass_squared1 (prt1) result (p2)
      type(prt_t), intent(in) :: prt1
```

```
    p2 = prt_get_msq (prt1)
  end function obs_mass_squared1
```

The signed invariant mass, which is the signed square root of the previous
observable.

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_signed_mass1 (prt1) result (m)
      type(prt_t), intent(in) :: prt1
      real(default) :: msq
      msq = prt_get_msq (prt1)
      m = sign (sqrt (abs (msq)), msq)
    end function obs_signed_mass1
```

The particle energy

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_energy1 (prt1) result (e)
      type(prt_t), intent(in) :: prt1
      e = energy (prt_get_momentum (prt1))
    end function obs_energy1
```

Particle momentum (components)

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_px1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = vector4_get_component (prt_get_momentum (prt1), 1)
    end function obs_px1

    real(default) function obs_py1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = vector4_get_component (prt_get_momentum (prt1), 2)
    end function obs_py1

    real(default) function obs_pz1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = vector4_get_component (prt_get_momentum (prt1), 3)
    end function obs_pz1

    real(default) function obs_p1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = space_part_norm (prt_get_momentum (prt1))
    end function obs_p1

    real(default) function obs_pl1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = longitudinal_part (prt_get_momentum (prt1))
    end function obs_pl1

    real(default) function obs_pt1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = transverse_part (prt_get_momentum (prt1))
    end function obs_pt1
```

Polar and azimuthal angle (lab frame).

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_theta1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = polar_angle (prt_get_momentum (prt1))
    end function obs_theta1

    real(default) function obs_phi1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = azimuthal_angle (prt_get_momentum (prt1))
    end function obs_phi1
```

Rapidity and pseudorapidity

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_rap1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = rapidity (prt_get_momentum (prt1))
    end function obs_rap1

    real(default) function obs_eta1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = pseudorapidity (prt_get_momentum (prt1))
    end function obs_eta1
```

Meaningless: Polar angle in the rest frame of the 2nd argument.

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_theta_rf1 (prt1) result (dist)
      type(prt_t), intent(in) :: prt1
      call msg_fatal (" 'Theta_RF' is undefined as unary observable")
      dist = 0
    end function obs_theta_rf1
```

Meaningless: Distance on the η-ϕ cylinder.

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_dist1 (prt1) result (dist)
      type(prt_t), intent(in) :: prt1
      call msg_fatal (" 'Dist' is undefined as unary observable")
      dist = 0
    end function obs_dist1
```

Integer-valued binary observables

These observables are meaningless as binary functions.

⟨*Variables: procedures*⟩+≡
```
    integer function obs_pdg2 (prt1, prt2) result (pdg)
      type(prt_t), intent(in) :: prt1, prt2
      call msg_fatal (" PDG_Code is undefined as binary observable")
      pdg = 0
    end function obs_pdg2
```

```
      integer function obs_helicity2 (prt1, prt2) result (h)
        type(prt_t), intent(in) :: prt1, prt2
        call msg_fatal (" Helicity is undefined as binary observable")
        h = 0
      end function obs_helicity2
```

Real-valued binary observables

The invariant mass squared, obtained from the separately stored value.

⟨*Variables: procedures*⟩+≡
```
      real(default) function obs_mass_squared2 (prt1, prt2) result (p2)
        type(prt_t), intent(in) :: prt1, prt2
        type(prt_t) :: prt
        call prt_init_combine (prt, prt1, prt2)
        p2 = prt_get_msq (prt)
      end function obs_mass_squared2
```

The signed invariant mass, which is the signed square root of the previous observable.

⟨*Variables: procedures*⟩+≡
```
      real(default) function obs_signed_mass2 (prt1, prt2) result (m)
        type(prt_t), intent(in) :: prt1, prt2
        type(prt_t) :: prt
        real(default) :: msq
        call prt_init_combine (prt, prt1, prt2)
        msq = prt_get_msq (prt)
        m = sign (sqrt (abs (msq)), msq)
      end function obs_signed_mass2
```

The particle energy

⟨*Variables: procedures*⟩+≡
```
      real(default) function obs_energy2 (prt1, prt2) result (e)
        type(prt_t), intent(in) :: prt1, prt2
        type(prt_t) :: prt
        call prt_init_combine (prt, prt1, prt2)
        e = energy (prt_get_momentum (prt))
      end function obs_energy2
```

Particle momentum (components)

⟨*Variables: procedures*⟩+≡
```
      real(default) function obs_px2 (prt1, prt2) result (p)
        type(prt_t), intent(in) :: prt1, prt2
        type(prt_t) :: prt
        call prt_init_combine (prt, prt1, prt2)
        p = vector4_get_component (prt_get_momentum (prt), 1)
      end function obs_px2

      real(default) function obs_py2 (prt1, prt2) result (p)
        type(prt_t), intent(in) :: prt1, prt2
        type(prt_t) :: prt
        call prt_init_combine (prt, prt1, prt2)
```

```
      p = vector4_get_component (prt_get_momentum (prt), 2)
    end function obs_py2

    real(default) function obs_pz2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
      type(prt_t) :: prt
      call prt_init_combine (prt, prt1, prt2)
      p = vector4_get_component (prt_get_momentum (prt), 3)
    end function obs_pz2

    real(default) function obs_p2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
      type(prt_t) :: prt
      call prt_init_combine (prt, prt1, prt2)
      p = space_part_norm (prt_get_momentum (prt))
    end function obs_p2

    real(default) function obs_pl2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
      type(prt_t) :: prt
      call prt_init_combine (prt, prt1, prt2)
      p = longitudinal_part (prt_get_momentum (prt))
    end function obs_pl2

    real(default) function obs_pt2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
      type(prt_t) :: prt
      call prt_init_combine (prt, prt1, prt2)
      p = transverse_part (prt_get_momentum (prt))
    end function obs_pt2
```

Enclosed angle and azimuthal distance (lab frame).

⟨*Variables: procedures*⟩+≡

```
    real(default) function obs_theta2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
      p = enclosed_angle (prt_get_momentum (prt1), prt_get_momentum (prt2))
    end function obs_theta2

    real(default) function obs_phi2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
      type(prt_t) :: prt
      call prt_init_combine (prt, prt1, prt2)
      p = azimuthal_distance (prt_get_momentum (prt1), prt_get_momentum (prt2))
    end function obs_phi2
```

Rapidity and pseudorapidity distance

⟨*Variables: procedures*⟩+≡

```
    real(default) function obs_rap2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
      p = rapidity_distance &
           (prt_get_momentum (prt1), prt_get_momentum (prt2))
    end function obs_rap2
```

```
real(default) function obs_eta2 (prt1, prt2) result (p)
  type(prt_t), intent(in) :: prt1, prt2
  type(prt_t) :: prt
  call prt_init_combine (prt, prt1, prt2)
  p = pseudorapidity_distance &
       (prt_get_momentum (prt1), prt_get_momentum (prt2))
end function obs_eta2
```

Polar angle in the rest frame of the 2nd argument.

⟨*Variables: procedures*⟩+≡
```
real(default) function obs_theta_rf2 (prt1, prt2) result (theta)
  type(prt_t), intent(in) :: prt1, prt2
  theta = enclosed_angle_rest_frame &
       (prt_get_momentum (prt1), prt_get_momentum (prt2))
end function obs_theta_rf2
```

Distance on the η-ϕ cylinder.

⟨*Variables: procedures*⟩+≡
```
real(default) function obs_dist2 (prt1, prt2) result (dist)
  type(prt_t), intent(in) :: prt1, prt2
  dist = eta_phi_distance &
       (prt_get_momentum (prt1), prt_get_momentum (prt2))
end function obs_dist2
```

5.4.9 API for variable lists

Set a new value. If the variable holds a pointer, this pointer is followed, e.g., a model parameter is actually set. If `ignore` is set, do nothing if the variable does not exist. If `verbose` is set, echo the new value.

⟨*Variables: public*⟩+≡
```
public :: var_list_set_log
public :: var_list_set_int
public :: var_list_set_real
public :: var_list_set_cmplx
public :: var_list_set_prt_list
public :: var_list_set_pdg_array
public :: var_list_set_string
```

⟨*Variables: procedures*⟩+≡
```
subroutine var_list_set_log &
     (var_list, name, lval, is_known, ignore, verbose, model_name)
  type(var_list_t), intent(inout), target :: var_list
  type(string_t), intent(in) :: name
  logical, intent(in) :: lval
  logical, intent(in) :: is_known
  logical, intent(in), optional :: ignore, verbose
  type(string_t), intent(in), optional :: model_name
  type(var_entry_t), pointer :: var
  var => var_list_get_var_ptr (var_list, name, V_LOG)
  if (associated (var)) then
     if (.not. var_entry_is_locked (var)) then
        select case (var%type)
```

```
              case (V_LOG)
                 call var_entry_set_log (var, lval, is_known, verbose, model_name)
              case default
                 call var_mismatch_error (name)
              end select
           else
              call var_locked_error (name)
           end if
        else
           call var_missing_error (name, ignore)
        end if
     end subroutine var_list_set_log

     subroutine var_list_set_int &
          (var_list, name, ival, is_known, ignore, verbose, model_name)
       type(var_list_t), intent(inout), target :: var_list
       type(string_t), intent(in) :: name
       integer, intent(in) :: ival
       logical, intent(in) :: is_known
       logical, intent(in), optional :: ignore, verbose
       type(string_t), intent(in), optional :: model_name
       type(var_entry_t), pointer :: var
       var => var_list_get_var_ptr (var_list, name, V_INT)
       if (associated (var)) then
          if (.not. var_entry_is_locked (var)) then
             select case (var%type)
             case (V_INT)
                call var_entry_set_int (var, ival, is_known, verbose, model_name)
             case default
                call var_mismatch_error (name)
             end select
          else
             call var_locked_error (name)
          end if
       else
          call var_missing_error (name, ignore)
       end if
     end subroutine var_list_set_int

     subroutine var_list_set_real &
          (var_list, name, rval, is_known, ignore, verbose, model_name)
       type(var_list_t), intent(inout), target :: var_list
       type(string_t), intent(in) :: name
       real(default), intent(in) :: rval
       logical, intent(in) :: is_known
       logical, intent(in), optional :: ignore, verbose
       type(string_t), intent(in), optional :: model_name
       type(var_entry_t), pointer :: var
       var => var_list_get_var_ptr (var_list, name, V_REAL)
       if (associated (var)) then
          if (.not. var_entry_is_locked (var)) then
             select case (var%type)
             case (V_REAL)
                call var_entry_set_real (var, rval, is_known, verbose, model_name)
```

```
            case default
                call var_mismatch_error (name)
            end select
        else
            call var_locked_error (name)
        end if
    else
        call var_missing_error (name, ignore)
    end if
end subroutine var_list_set_real

subroutine var_list_set_cmplx &
      (var_list, name, cval, is_known, ignore, verbose, model_name)
    type(var_list_t), intent(inout), target :: var_list
    type(string_t), intent(in) :: name
    complex(default), intent(in) :: cval
    logical, intent(in) :: is_known
    logical, intent(in), optional :: ignore, verbose
    type(string_t), intent(in), optional :: model_name
    type(var_entry_t), pointer :: var
    var => var_list_get_var_ptr (var_list, name, V_CMPLX)
    if (associated (var)) then
        if (.not. var_entry_is_locked (var)) then
            select case (var%type)
            case (V_CMPLX)
                call var_entry_set_cmplx (var, cval, is_known, verbose, model_name)
            case default
                call var_mismatch_error (name)
            end select
        else
            call var_locked_error (name)
        end if
    else
        call var_missing_error (name, ignore)
    end if
end subroutine var_list_set_cmplx

subroutine var_list_set_pdg_array &
      (var_list, name, aval, is_known, ignore, verbose, model_name)
    type(var_list_t), intent(inout), target :: var_list
    type(string_t), intent(in) :: name
    type(pdg_array_t), intent(in) :: aval
    logical, intent(in) :: is_known
    logical, intent(in), optional :: ignore, verbose
    type(string_t), intent(in), optional :: model_name
    type(var_entry_t), pointer :: var
    var => var_list_get_var_ptr (var_list, name, V_PDG)
    if (associated (var)) then
        if (.not. var_entry_is_locked (var)) then
            select case (var%type)
            case (V_PDG)
                call var_entry_set_pdg_array &
                        (var, aval, is_known, verbose, model_name)
            case default
```

```fortran
           call var_mismatch_error (name)
        end select
     else
        call var_locked_error (name)
     end if
  else
     call var_missing_error (name, ignore)
  end if
end subroutine var_list_set_pdg_array

subroutine var_list_set_prt_list &
     (var_list, name, pval, is_known, ignore, verbose, model_name)
  type(var_list_t), intent(inout), target :: var_list
  type(string_t), intent(in) :: name
  type(prt_list_t), intent(in) :: pval
  logical, intent(in) :: is_known
  logical, intent(in), optional :: ignore, verbose
  type(string_t), intent(in), optional :: model_name
  type(var_entry_t), pointer :: var
  var => var_list_get_var_ptr (var_list, name, V_PTL)
  if (associated (var)) then
     if (.not. var_entry_is_locked (var)) then
        select case (var%type)
        case (V_PTL)
           call var_entry_set_prt_list &
                (var, pval, is_known, verbose, model_name)
        case default
           call var_mismatch_error (name)
        end select
     else
        call var_locked_error (name)
     end if
  else
     call var_missing_error (name, ignore)
  end if
end subroutine var_list_set_prt_list

subroutine var_list_set_string &
     (var_list, name, sval, is_known, ignore, verbose, model_name)
  type(var_list_t), intent(inout), target :: var_list
  type(string_t), intent(in) :: name
  type(string_t), intent(in) :: sval
  logical, intent(in) :: is_known
  logical, intent(in), optional :: ignore, verbose
  type(string_t), intent(in), optional :: model_name
  type(var_entry_t), pointer :: var
  var => var_list_get_var_ptr (var_list, name, V_STR)
  if (associated (var)) then
     if (.not. var_entry_is_locked (var)) then
        select case (var%type)
        case (V_STR)
           call var_entry_set_string &
                (var, sval, is_known, verbose, model_name)
        case default
```

```
          call var_mismatch_error (name)
        end select
     else
        call var_locked_error (name)
     end if
  else
     call var_missing_error (name, ignore)
  end if
end subroutine var_list_set_string

subroutine var_mismatch_error (name)
  type(string_t), intent(in) :: name
  call msg_fatal ("Type mismatch for variable '" // char (name) // "'")
end subroutine var_mismatch_error

subroutine var_locked_error (name)
  type(string_t), intent(in) :: name
  call msg_error ("Variable '" // char (name) // "' is not user-definable")
end subroutine var_locked_error

subroutine var_missing_error (name, ignore)
  type(string_t), intent(in) :: name
  logical, intent(in), optional :: ignore
  logical :: error
  if (present (ignore)) then
     error = .not. ignore
  else
     error = .true.
  end if
  if (error) then
     call msg_fatal ("Variable '" // char (name) // "' has not been declared")
  end if
end subroutine var_missing_error
```

5.4.10 Linking model variables

The variable list of a model can be linked to the global variable list, so the
model variables become available. However, the model may change during the
execution of the command list, and this is not known at compile time. So, we
make a copy of all variables that can be modified by the user; this will include
all variables that are present in any of the models. At runtime, the linked list
and the pointers to it can be exchanged, but the global variables will stay.

  Append a single model variable to the local variable list. The pointer to the
original is not set (yet). Check first if it already exists; in that case, do nothing.

⟨*Variables: public*⟩+≡
```
    public :: var_list_init_copy
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_init_copy (var_list, model_var)
      type(var_list_t), intent(inout), target :: var_list
      type(var_entry_t), intent(in), target :: model_var
      type(var_entry_t), pointer :: var
```

```
      if (.not. var_list_exists &
          (var_list, model_var%name, follow_link = .false.)) then
         allocate (var)
         call var_entry_init_copy (var, model_var)
         call var_list_append (var_list, var)
      end if
   end subroutine var_list_init_copy
```

Append all model variables, or reuse (unset) them if they already exist. If `derived_only` is set, copy only derived parameters; these are real parameters that are locked. How should we (should we at all?) generalize this to complex parameters?

⟨*Variables: public*⟩+≡
```
   public :: var_list_init_copies
```

⟨*Variables: procedures*⟩+≡
```
   subroutine var_list_init_copies (var_list, model_vars, derived_only)
     type(var_list_t), intent(inout), target :: var_list
     type(var_list_t), intent(in) :: model_vars
     logical, intent(in), optional :: derived_only
     type(var_entry_t), pointer :: model_var, var
     type(string_t) :: name
     logical :: copy_all, locked, derived
     integer :: type
     copy_all = .true.
     if (present (derived_only))  copy_all = .not. derived_only
     model_var => model_vars%first
     do while (associated (model_var))
        name = var_entry_get_name (model_var)
        type = var_entry_get_type (model_var)
        locked = var_entry_is_locked (model_var)
        derived = type == V_REAL .and. locked
        if (copy_all .or. derived) then
           var => var_list_get_var_ptr &
                (var_list, name, type, follow_link = .false.)
           if (associated (var)) then
              call var_entry_undefine (var)
           else
              allocate (var)
              call var_entry_init_copy (var, model_var)
              call var_list_append (var_list, var)
           end if
        end if
        model_var => model_var%next
     end do
   end subroutine var_list_init_copies
```

Clear all previously allocated pointers to some model:

⟨*Variables: procedures*⟩+≡
```
   subroutine var_list_clear_original_pointers (var_list)
     type(var_list_t), intent(inout) :: var_list
     type(var_entry_t), pointer :: var
     var => var_list%first
```

```
      do while (associated (var))
         call var_entry_clear_original_pointer (var)
         var => var%next
      end do
   end subroutine var_list_clear_original_pointers
```

Assign the pointer to the original for a single variable.
⟨*Variables: public*⟩+≡
```
   public :: var_list_set_original_pointer
```

⟨*Variables: procedures*⟩+≡
```
   subroutine var_list_set_original_pointer (var_list, name, model_vars)
     type(var_list_t), intent(inout) :: var_list
     type(string_t), intent(in) :: name
     type(var_list_t), intent(in), target :: model_vars
     type(var_entry_t), pointer :: var, model_var
     integer :: type
     model_var => var_list_get_var_ptr (model_vars, name)
     if (associated (model_var)) then
        type = var_entry_get_type (model_var)
        var => var_list_get_var_ptr (var_list, name, type, follow_link=.false.)
        if (associated (var)) then
           call var_entry_set_original_pointer (var, model_var)
        end if
     end if
   end subroutine var_list_set_original_pointer
```

Assign pointers to the originals for all variables in the model variable list.
⟨*Variables: procedures*⟩+≡
```
   subroutine var_list_set_original_pointers (var_list, model_vars)
     type(var_list_t), intent(inout) :: var_list
     type(var_list_t), intent(in), target :: model_vars
     type(var_entry_t), pointer :: var, model_var
     type(string_t) :: name
     integer :: type
     model_var => model_vars%first
     do while (associated (model_var))
        name = var_entry_get_name (model_var)
        type = var_entry_get_type (model_var)
        var => var_list_get_var_ptr (var_list, name, type, follow_link=.false.)
        if (associated (var)) then
           call var_entry_set_original_pointer (var, model_var)
        end if
        model_var => model_var%next
     end do
   end subroutine var_list_set_original_pointers
```

Synchronize the local variable list with the model variable list (which is accessed by the pointers assigned in the previous subroutine).

If `reset_pointers` is unset, do not reset pointers but just update values where a variable is a copy. Resetting pointers is done only in the local variable list.

⟨*Variables: public*⟩+≡

```
      public :: var_list_synchronize
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_synchronize (var_list, model_vars, reset_pointers)
      type(var_list_t), intent(inout) :: var_list
      type(var_list_t), intent(in), target :: model_vars
      logical, intent(in), optional :: reset_pointers
      type(var_entry_t), pointer :: var
      if (present (reset_pointers)) then
         if (reset_pointers) then
            call var_list_clear_original_pointers (var_list)
            call var_list_set_original_pointers (var_list, model_vars)
         end if
      end if
      var => var_list%first
      do while (associated (var))
         call var_entry_synchronize (var)
         var => var%next
      end do
    end subroutine var_list_synchronize
```

This is the inverse operation: synchronize the model variable list with the current variable list. This is necessary after discarding a local variable list.

⟨*Variables: public*⟩+≡
```
      public :: var_list_restore
```

⟨*Variables: procedures*⟩+≡
```
    recursive subroutine var_list_restore (var_list)
      type(var_list_t), intent(inout) :: var_list
      type(var_entry_t), pointer :: var
      var => var_list%first
      do while (associated (var))
         call var_entry_restore (var)
         var => var%next
      end do
    end subroutine var_list_restore
```

Make a deep copy of a variable list. The copy does not contain any pointer variables. Clear the original pointer after use, since the original may be lost when the copy is in use.

⟨*Variables: public*⟩+≡
```
      public :: var_list_init_snapshot
```

⟨*Variables: procedures*⟩+≡
```
    recursive subroutine var_list_init_snapshot (var_list, vars_in, follow_link)
      type(var_list_t), intent(out) :: var_list
      type(var_list_t), intent(in) :: vars_in
      logical, intent(in), optional :: follow_link
      type(var_entry_t), pointer :: var, var_in
      type(var_list_t), pointer :: var_list_next
      logical :: rec
      rec = .true.;  if (present (follow_link))  rec = follow_link
      var_in => vars_in%first
      do while (associated (var_in))
```

```
        allocate (var)
        call var_entry_init_copy (var, var_in)
        call var_entry_set_original_pointer (var, var_in)
        call var_entry_synchronize (var)
        call var_entry_clear_original_pointer (var)
        call var_list_append (var_list, var)
        var_in => var_in%next
      end do
      if (rec .and. associated (vars_in%next)) then
        allocate (var_list_next)
        call var_list_init_snapshot (var_list_next, vars_in%next)
        call var_list_link (var_list, var_list_next)
      end if
    end subroutine var_list_init_snapshot
```

Check if a user variable can be set. The `new` flag is set if the user variable has an explicit declaration. If an error occurs, return `V_NONE` as variable type.

Also determine the actual type of generic numerical variables, which enter the procedure with type `V_NONE`.

⟨*Variables: public*⟩+≡
```
    public :: var_list_check_user_var
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_check_user_var (var_list, name, type, new)
      type(var_list_t), intent(in), target :: var_list
      type(string_t), intent(in) :: name
      integer, intent(inout) :: type
      logical, intent(in) :: new
      type(var_entry_t), pointer :: var
      if (string_is_observable_id (name)) then
        call msg_error ("Variable name '" // char (name) &
             // "' is reserved for an observable")
        type = V_NONE
        return
      end if
      var => var_list_get_var_ptr (var_list, name)
      if (associated (var)) then
        if (type == V_NONE) then
          type = var_entry_get_type (var)
        end if
        if (var_entry_is_locked (var)) then
          call msg_error ("Variable '" // char (name) &
               // "' is not user-definable")
          type = V_NONE
          return
        else if (new) then
          if (var_entry_is_intrinsic (var)) then
            call msg_error ("Intrinsic variable '" &
                 // char (name) // "' redeclared")
            type = V_NONE
            return
          end if
          if (var_entry_get_type (var) /= type) then
            call msg_error ("Variable '" // char (name) // "' " &
```

```
              // "redeclared with different type")
           type = V_NONE
           return
        end if
     end if
  else
     if (.not. new) then
        call msg_error ("Variable '" // char (name) // "' " &
           // "set without declaration")
        type = V_NONE
        return
     end if
  end if
end subroutine var_list_check_user_var
```

5.5   Expressions

In this module we define the structures needed to parse a user-defined expression,
to compile it into an evaluation tree, and to evaluate it.

We have two flavors of expressions: one with particles and one without
particles. The latter version is used for defining cut/selection criteria and for
online analysis.

⟨expressions.f90⟩≡
  ⟨*File header*⟩

```
module expressions
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
```
  use constants !NODEP!
```
  ⟨*Use file utils*⟩
```
  use diagnostics !NODEP!
  use lorentz !NODEP!
  use formats
  use sorting
  use ifiles
  use lexers
  use syntax_rules
  use parser
  use analysis
  use pdg_arrays
  use prt_lists
  use variables
```

  ⟨*Standard module head*⟩

  ⟨*Expressions: public*⟩

  ⟨*Expressions: types*⟩

  ⟨*Expressions: interfaces*⟩

⟨*Expressions: variables*⟩

```
contains
```

⟨*Expressions: procedures*⟩

```
end module expressions
```

5.5.1   Tree nodes

The evaluation tree consists of branch nodes (unary and binary) and of leaf nodes, originating from a common root. The node object should be polymorphic. For the time being, polymorphism is emulated here. This means that we have to maintain all possibilities that the node may hold, including associated procedures as pointers.

The following parameter values characterize the node. Unary and binary operators have sub-nodes. The other are leaf nodes. Possible leafs are literal constants or named-parameter references.

⟨*Expressions: types*⟩≡
```
    integer, parameter :: EN_UNKNOWN = 0, EN_UNARY = 1, EN_BINARY = 2
    integer, parameter :: EN_CONSTANT = 3, EN_VARIABLE = 4
    integer, parameter :: EN_CONDITIONAL = 5, EN_BLOCK = 6
    integer, parameter :: EN_RECORD_CMD = 7
    integer, parameter :: EN_OBS1_INT = 11, EN_OBS2_INT = 12
    integer, parameter :: EN_OBS1_REAL = 21, EN_OBS2_REAL = 22
    integer, parameter :: EN_PRT_FUN_UNARY = 101, EN_PRT_FUN_BINARY = 102
    integer, parameter :: EN_EVAL_FUN_UNARY = 111, EN_EVAL_FUN_BINARY = 112
    integer, parameter :: EN_LOG_FUN_UNARY = 121, EN_LOG_FUN_BINARY = 122
    integer, parameter :: EN_INT_FUN_UNARY = 131, EN_INT_FUN_BINARY = 132
    integer, parameter :: EN_FORMAT_STR = 141
```

⟨*Expressions: types*⟩+≡
```
    type :: eval_node_t
       private
       type(string_t) :: tag
       integer :: type = EN_UNKNOWN
       integer :: result_type = V_NONE
       type(var_list_t), pointer :: var_list => null ()
       type(string_t) :: var_name
       logical, pointer :: value_is_known => null ()
       logical,          pointer :: lval => null ()
       integer,          pointer :: ival => null ()
       real(default),    pointer :: rval => null ()
       complex(default), pointer :: cval => null ()
       type(prt_list_t), pointer :: pval => null ()
       type(pdg_array_t), pointer :: aval => null ()
       type(string_t),   pointer :: sval => null ()
       type(eval_node_t), pointer :: arg0 => null ()
       type(eval_node_t), pointer :: arg1 => null ()
       type(eval_node_t), pointer :: arg2 => null ()
       procedure(obs_unary_int),   nopass, pointer :: obs1_int  => null ()
```

```
      procedure(obs_unary_real),  nopass, pointer :: obs1_real => null ()
      procedure(obs_binary_int),  nopass, pointer :: obs2_int  => null ()
      procedure(obs_binary_real), nopass, pointer :: obs2_real => null ()
      integer, pointer :: prt_type => null ()
      integer, pointer :: index => null ()
      real(default), pointer :: tolerance => null ()
      type(prt_t), pointer :: prt1 => null ()
      type(prt_t), pointer :: prt2 => null ()
      procedure(unary_log),  nopass, pointer :: op1_log  => null ()
      procedure(unary_int),  nopass, pointer :: op1_int  => null ()
      procedure(unary_real), nopass, pointer :: op1_real => null ()
      procedure(unary_cmplx), nopass, pointer :: op1_cmplx => null ()
      procedure(unary_pdg),  nopass, pointer :: op1_pdg  => null ()
      procedure(unary_ptl),  nopass, pointer :: op1_ptl  => null ()
      procedure(unary_str),  nopass, pointer :: op1_str  => null ()
      procedure(unary_cut),  nopass, pointer :: op1_cut  => null ()
      procedure(unary_num),  nopass, pointer :: op1_num  => null ()
      procedure(binary_log),  nopass, pointer :: op2_log  => null ()
      procedure(binary_int),  nopass, pointer :: op2_int  => null ()
      procedure(binary_real), nopass, pointer :: op2_real => null ()
      procedure(binary_cmplx), nopass, pointer :: op2_cmplx => null ()
      procedure(binary_pdg),  nopass, pointer :: op2_pdg  => null ()
      procedure(binary_ptl),  nopass, pointer :: op2_ptl  => null ()
      procedure(binary_str),  nopass, pointer :: op2_str  => null ()
      procedure(binary_cut),  nopass, pointer :: op2_cut  => null ()
      procedure(binary_num),  nopass, pointer :: op2_num  => null ()
   end type eval_node_t
```

Finalize a node recursively. Allocated constants are deleted, pointers are ignored.

⟨*Expressions: procedures*⟩≡
```
   recursive subroutine eval_node_final_rec (node)
     type(eval_node_t), intent(inout) :: node
     select case (node%type)
     case (EN_UNARY)
        call eval_node_final_rec (node%arg1)
     case (EN_BINARY)
        call eval_node_final_rec (node%arg1)
        call eval_node_final_rec (node%arg2)
     case (EN_CONDITIONAL)
        call eval_node_final_rec (node%arg0)
        call eval_node_final_rec (node%arg1)
        call eval_node_final_rec (node%arg2)
     case (EN_BLOCK)
        call eval_node_final_rec (node%arg0)
        call eval_node_final_rec (node%arg1)
     case (EN_PRT_FUN_UNARY, EN_EVAL_FUN_UNARY, &
           EN_LOG_FUN_UNARY, EN_INT_FUN_UNARY)
        if (associated (node%arg0))  call eval_node_final_rec (node%arg0)
        call eval_node_final_rec (node%arg1)
        deallocate (node%index)
        deallocate (node%prt1)
     case (EN_PRT_FUN_BINARY, EN_EVAL_FUN_BINARY, &
```

```
          EN_LOG_FUN_BINARY, EN_INT_FUN_BINARY)
      if (associated (node%arg0))  call eval_node_final_rec (node%arg0)
      call eval_node_final_rec (node%arg1)
      call eval_node_final_rec (node%arg2)
      deallocate (node%index)
      deallocate (node%prt1)
      deallocate (node%prt2)
   case (EN_FORMAT_STR)
      if (associated (node%arg0))  call eval_node_final_rec (node%arg0)
      if (associated (node%arg1))  call eval_node_final_rec (node%arg1)
      deallocate (node%ival)
   end select
   select case (node%type)
   case (EN_UNARY, EN_BINARY, EN_CONDITIONAL, EN_CONSTANT, EN_BLOCK, &
         EN_PRT_FUN_UNARY, EN_PRT_FUN_BINARY, &
         EN_EVAL_FUN_UNARY, EN_EVAL_FUN_BINARY, &
         EN_LOG_FUN_UNARY, EN_LOG_FUN_BINARY, &
         EN_INT_FUN_UNARY, EN_INT_FUN_BINARY, &
         EN_FORMAT_STR)
      select case (node%result_type)
      case (V_LOG);  deallocate (node%lval)
      case (V_INT);  deallocate (node%ival)
      case (V_REAL); deallocate (node%rval)
      case (V_CMPLX); deallocate (node%cval)
      case (V_PTL);  deallocate (node%pval)
      case (V_PDG);  deallocate (node%aval)
      case (V_STR);  deallocate (node%sval)
      end select
      deallocate (node%value_is_known)
   end select
 end subroutine eval_node_final_rec
```

Leaf nodes

Initialize a leaf node with a literal constant.

⟨*Expressions: procedures*⟩+≡

```
    subroutine eval_node_init_log (node, lval)
      type(eval_node_t), intent(out) :: node
      logical, intent(in) :: lval
      node%type = EN_CONSTANT
      node%result_type = V_LOG
      allocate (node%lval, node%value_is_known)
      node%lval = lval
      node%value_is_known = .true.
    end subroutine eval_node_init_log

    subroutine eval_node_init_int (node, ival)
      type(eval_node_t), intent(out) :: node
      integer, intent(in) :: ival
      node%type = EN_CONSTANT
      node%result_type = V_INT
      allocate (node%ival, node%value_is_known)
      node%ival = ival
```

```
    node%value_is_known = .true.
  end subroutine eval_node_init_int

  subroutine eval_node_init_real (node, rval)
    type(eval_node_t), intent(out) :: node
    real(default), intent(in) :: rval
    node%type = EN_CONSTANT
    node%result_type = V_REAL
    allocate (node%rval, node%value_is_known)
    node%rval = rval
    node%value_is_known = .true.
  end subroutine eval_node_init_real

  subroutine eval_node_init_cmplx (node, cval)
    type(eval_node_t), intent(out) :: node
    complex(default), intent(in) :: cval
    node%type = EN_CONSTANT
    node%result_type = V_CMPLX
    allocate (node%cval, node%value_is_known)
    node%cval = cval
    node%value_is_known = .true.
  end subroutine eval_node_init_cmplx

  subroutine eval_node_init_prt_list (node, pval)
    type(eval_node_t), intent(out) :: node
    type(prt_list_t), intent(in) :: pval
    node%type = EN_CONSTANT
    node%result_type = V_PTL
    allocate (node%pval, node%value_is_known)
    node%pval = pval
    node%value_is_known = .true.
  end subroutine eval_node_init_prt_list

  subroutine eval_node_init_pdg_array (node, aval)
    type(eval_node_t), intent(out) :: node
    type(pdg_array_t), intent(in) :: aval
    node%type = EN_CONSTANT
    node%result_type = V_PDG
    allocate (node%aval, node%value_is_known)
    node%aval = aval
    node%value_is_known = .true.
  end subroutine eval_node_init_pdg_array

  subroutine eval_node_init_string (node, sval)
    type(eval_node_t), intent(out) :: node
    type(string_t), intent(in) :: sval
    node%type = EN_CONSTANT
    node%result_type = V_STR
    allocate (node%sval, node%value_is_known)
    node%sval = sval
    node%value_is_known = .true.
  end subroutine eval_node_init_string
```

Initialize a leaf node with a pointer to a named parameter

⟨Expressions: procedures⟩+≡

```
subroutine eval_node_init_log_ptr (node, name, lval, is_known)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  logical, intent(in), target :: lval
  logical, intent(in), target :: is_known
  node%type = EN_VARIABLE
  node%tag = name
  node%result_type = V_LOG
  node%lval => lval
  node%value_is_known => is_known
end subroutine eval_node_init_log_ptr

subroutine eval_node_init_int_ptr (node, name, ival, is_known)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  integer, intent(in), target :: ival
  logical, intent(in), target :: is_known
  node%type = EN_VARIABLE
  node%tag = name
  node%result_type = V_INT
  node%ival => ival
  node%value_is_known => is_known
end subroutine eval_node_init_int_ptr

subroutine eval_node_init_real_ptr (node, name, rval, is_known)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  real(default), intent(in), target :: rval
  logical, intent(in), target :: is_known
  node%type = EN_VARIABLE
  node%tag = name
  node%result_type = V_REAL
  node%rval => rval
  node%value_is_known => is_known
end subroutine eval_node_init_real_ptr

subroutine eval_node_init_cmplx_ptr (node, name, cval, is_known)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  complex(default), intent(in), target :: cval
  logical, intent(in), target :: is_known
  node%type = EN_VARIABLE
  node%tag = name
  node%result_type = V_CMPLX
  node%cval => cval
  node%value_is_known => is_known
end subroutine eval_node_init_cmplx_ptr

subroutine eval_node_init_prt_list_ptr (node, name, pval, is_known)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  type(prt_list_t), intent(in), target :: pval
  logical, intent(in), target :: is_known
```

```
      node%type = EN_VARIABLE
      node%tag = name
      node%result_type = V_PTL
      node%pval => pval
      node%value_is_known => is_known
    end subroutine eval_node_init_prt_list_ptr

    subroutine eval_node_init_pdg_array_ptr (node, name, aval, is_known)
      type(eval_node_t), intent(out) :: node
      type(string_t), intent(in) :: name
      type(pdg_array_t), intent(in), target :: aval
      logical, intent(in), target :: is_known
      node%type = EN_VARIABLE
      node%tag = name
      node%result_type = V_PDG
      node%aval => aval
      node%value_is_known => is_known
    end subroutine eval_node_init_pdg_array_ptr

    subroutine eval_node_init_string_ptr (node, name, sval, is_known)
      type(eval_node_t), intent(out) :: node
      type(string_t), intent(in) :: name
      type(string_t), intent(in), target :: sval
      logical, intent(in), target :: is_known
      node%type = EN_VARIABLE
      node%tag = name
      node%result_type = V_STR
      node%sval => sval
      node%value_is_known => is_known
    end subroutine eval_node_init_string_ptr
```

Initialize a leaf node with an observable. This includes a procedures pointer.
The input is a variable entry from the stack which holds the necessary informa-
tion.

⟨*Expressions: procedures*⟩+≡

```
    subroutine eval_node_init_obs (node, var)
      type(eval_node_t), intent(out) :: node
      type(var_entry_t), intent(in), target :: var
      node%tag = var_entry_get_name (var)
      select case (var_entry_get_type (var))
      case (V_OBS1_INT)
         node%type = EN_OBS1_INT
         call var_entry_assign_obs1_int_ptr (node%obs1_int, var)
      case (V_OBS2_INT)
         node%type = EN_OBS2_INT
         call var_entry_assign_obs2_int_ptr (node%obs2_int, var)
      case (V_OBS1_REAL)
         node%type = EN_OBS1_REAL
         call var_entry_assign_obs1_real_ptr (node%obs1_real, var)
      case (V_OBS2_REAL)
         node%type = EN_OBS2_REAL
         call var_entry_assign_obs2_real_ptr (node%obs2_real, var)
      end select
```

```
        select case (var_entry_get_type (var))
        case (V_OBS1_INT, V_OBS2_INT)
           node%result_type = V_INT
           allocate (node%ival, node%value_is_known)
           node%value_is_known = .false.
        case (V_OBS1_REAL, V_OBS2_REAL)
           node%result_type = V_REAL
           allocate (node%rval, node%value_is_known)
           node%value_is_known = .false.
        end select
        select case (var_entry_get_type (var))
        case (V_OBS1_INT, V_OBS1_REAL)
           node%prt1 => var_entry_get_prt1_ptr (var)
        case (V_OBS2_INT, V_OBS2_REAL)
           node%prt1 => var_entry_get_prt1_ptr (var)
           node%prt2 => var_entry_get_prt2_ptr (var)
        end select
      end subroutine eval_node_init_obs
```

Branch nodes

Initialize a branch node, sub-nodes are given.

⟨*Expressions: procedures*⟩+≡

```
    subroutine eval_node_init_branch (node, tag, result_type, arg1, arg2)
      type(eval_node_t), intent(out) :: node
      type(string_t), intent(in) :: tag
      integer, intent(in) :: result_type
      type(eval_node_t), intent(in), target :: arg1
      type(eval_node_t), intent(in), target, optional :: arg2
      if (present (arg2)) then
         node%type = EN_BINARY
      else
         node%type = EN_UNARY
      end if
      node%tag = tag
      node%result_type = result_type
      call eval_node_allocate_value (node)
      node%arg1 => arg1
      if (present (arg2))  node%arg2 => arg2
    end subroutine eval_node_init_branch
```

Allocate the node value according to the result type. Type 'pdg array' is not listed since this is not a scalar.

⟨*Expressions: procedures*⟩+≡

```
    subroutine eval_node_allocate_value (node)
      type(eval_node_t), intent(inout) :: node
      select case (node%result_type)
      case (V_LOG);  allocate (node%lval)
      case (V_INT);  allocate (node%ival)
      case (V_REAL); allocate (node%rval)
      case (V_CMPLX); allocate (node%cval)
      case (V_PDG);  allocate (node%aval)
```

```
      case (V_PTL);  allocate (node%pval)
         call prt_list_init (node%pval)
      case (V_STR);  allocate (node%sval)
      end select
      allocate (node%value_is_known)
    end subroutine eval_node_allocate_value
```

Initialize a block node which contains, in addition to the expression to be evaluated, a variable definition. The result type is not yet assigned, because we can compile the enclosed expression only after the var list is set up.

Note that the node always allocates a new variable list and appends it to the current one. Thus, if the variable redefines an existing one, it only shadows it but does not reset it. Any side-effects are therefore absent and need not be undone outside the block.

If the flag new is set, a variable is (re)declared. This must not be done for intrinsic variables. Vice versa, if the variable is not existent, the new flag is required.

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_node_init_block (node, name, type, var_def, var_list)
      type(eval_node_t), intent(out), target :: node
      type(string_t), intent(in) :: name
      integer, intent(in) :: type
      type(eval_node_t), intent(in), target :: var_def
      type(var_list_t), intent(in), target :: var_list
      node%type = EN_BLOCK
      node%tag = "var_def"
      node%var_name = name
      node%arg1 => var_def
      allocate (node%var_list)
      call var_list_link (node%var_list, var_list)
      if (var_def%type == EN_CONSTANT) then
         select case (type)
         case (V_LOG)
            call var_list_append_log  (node%var_list, name, var_def%lval)
         case (V_INT)
            call var_list_append_int  (node%var_list, name, var_def%ival)
         case (V_REAL)
            call var_list_append_real (node%var_list, name, var_def%rval)
         case (V_CMPLX)
            call var_list_append_cmplx (node%var_list, name, var_def%cval)
         case (V_PDG)
            call var_list_append_pdg_array &
                 (node%var_list, name, var_def%aval)
         case (V_PTL)
            call var_list_append_prt_list &
                 (node%var_list, name, var_def%pval)
         case (V_STR)
            call var_list_append_string (node%var_list, name, var_def%sval)
         end select
      else
         select case (type)
         case (V_LOG);  call var_list_append_log_ptr &
                 (node%var_list, name, var_def%lval, var_def%value_is_known)
```

```
            case (V_INT);   call var_list_append_int_ptr &
                 (node%var_list, name, var_def%ival, var_def%value_is_known)
            case (V_REAL); call var_list_append_real_ptr &
                 (node%var_list, name, var_def%rval, var_def%value_is_known)
            case (V_CMPLX); call var_list_append_cmplx_ptr &
                 (node%var_list, name, var_def%cval, var_def%value_is_known)
            case (V_PDG);   call var_list_append_pdg_array_ptr &
                 (node%var_list, name, var_def%aval, var_def%value_is_known)
            case (V_PTL); call var_list_append_prt_list_ptr &
                 (node%var_list, name, var_def%pval, var_def%value_is_known)
            case (V_STR); call var_list_append_string_ptr &
                 (node%var_list, name, var_def%sval, var_def%value_is_known)
            end select
        end if
    end subroutine eval_node_init_block
```

Complete block initialization by assigning the expression to evaluate to `arg0`.

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_node_set_expr (node, arg, result_type)
      type(eval_node_t), intent(inout) :: node
      type(eval_node_t), intent(in), target :: arg
      integer, intent(in), optional :: result_type
      if (present (result_type)) then
         node%result_type = result_type
      else
         node%result_type = arg%result_type
      end if
      call eval_node_allocate_value (node)
      node%arg0 => arg
    end subroutine eval_node_set_expr
```

Initialize a conditional. There are three branches: the condition (evaluates to logical) and the two alternatives (evaluate both to the same arbitrary type).

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_node_init_conditional (node, result_type, cond, arg1, arg2)
      type(eval_node_t), intent(out) :: node
      integer, intent(in) :: result_type
      type(eval_node_t), intent(in), target :: cond, arg1, arg2
      node%type = EN_CONDITIONAL
      node%tag = "cond"
      node%result_type = result_type
      call eval_node_allocate_value (node)
      node%arg0 => cond
      node%arg1 => arg1
      node%arg2 => arg2
    end subroutine eval_node_init_conditional
```

Initialize a recording command (which evaluates to a logical constant). The first branch is the ID of the analysis object to be filled, the optional branches 1 and 2 are the values to be recorded.

   If the event-weight pointer is null, we record values with unit weight. Otherwise, we use the value pointed to as event weight.

⟨Expressions: procedures⟩+≡
```
subroutine eval_node_init_record_cmd (node, event_weight, id, arg1, arg2)
  type(eval_node_t), intent(out) :: node
  real(default), pointer :: event_weight
  type(eval_node_t), intent(in), target :: id
  type(eval_node_t), intent(in), optional, target :: arg1, arg2
  call eval_node_init_log (node, .true.)
  node%type = EN_RECORD_CMD
  node%rval => event_weight
  node%tag = "record_cmd"
  node%arg0 => id
  if (present (arg1)) then
     node%arg1 => arg1
     if (present (arg2)) then
        node%arg2 => arg2
     end if
  end if
end subroutine eval_node_init_record_cmd
```

Initialize a node for operations on particle lists. The particle lists (one or two) are inserted as `arg1` and `arg2`. We allocated particle pointers as temporaries for iterating over particle lists. The procedure pointer which holds the function to evaluate for the particle lists (e.g., combine, select) is also initialized.

⟨Expressions: procedures⟩+≡
```
subroutine eval_node_init_prt_fun_unary (node, arg1, name, proc)
  type(eval_node_t), intent(out) :: node
  type(eval_node_t), intent(in), target :: arg1
  type(string_t), intent(in) :: name
  procedure(unary_ptl) :: proc
  node%type = EN_PRT_FUN_UNARY
  node%tag = name
  node%result_type = V_PTL
  call eval_node_allocate_value (node)
  node%arg1 => arg1
  allocate (node%index)
  allocate (node%prt1)
  node%op1_ptl => proc
end subroutine eval_node_init_prt_fun_unary

subroutine eval_node_init_prt_fun_binary (node, arg1, arg2, name, proc)
  type(eval_node_t), intent(out) :: node
  type(eval_node_t), intent(in), target :: arg1, arg2
  type(string_t), intent(in) :: name
  procedure(binary_ptl) :: proc
  node%type = EN_PRT_FUN_BINARY
  node%tag = name
  node%result_type = V_PTL
  call eval_node_allocate_value (node)
  node%arg1 => arg1
  node%arg2 => arg2
  allocate (node%index)
  allocate (node%prt1)
  allocate (node%prt2)
```

```
      node%op2_ptl => proc
    end subroutine eval_node_init_prt_fun_binary
```

Similar, but for particle-list functions that evaluate to a real value.

⟨*Expressions: procedures*⟩+≡

```
    subroutine eval_node_init_eval_fun_unary (node, arg1, name)
      type(eval_node_t), intent(out) :: node
      type(eval_node_t), intent(in), target :: arg1
      type(string_t), intent(in) :: name
      node%type = EN_EVAL_FUN_UNARY
      node%tag = name
      node%result_type = V_REAL
      call eval_node_allocate_value (node)
      node%arg1 => arg1
      allocate (node%index)
      allocate (node%prt1)
    end subroutine eval_node_init_eval_fun_unary

    subroutine eval_node_init_eval_fun_binary (node, arg1, arg2, name)
      type(eval_node_t), intent(out) :: node
      type(eval_node_t), intent(in), target :: arg1, arg2
      type(string_t), intent(in) :: name
      node%type = EN_EVAL_FUN_BINARY
      node%tag = name
      node%result_type = V_REAL
      call eval_node_allocate_value (node)
      node%arg1 => arg1
      node%arg2 => arg2
      allocate (node%index)
      allocate (node%prt1)
      allocate (node%prt2)
    end subroutine eval_node_init_eval_fun_binary
```

These are for particle-list functions that evaluate to a logical value.

⟨*Expressions: procedures*⟩+≡

```
    subroutine eval_node_init_log_fun_unary (node, arg1, name, proc)
      type(eval_node_t), intent(out) :: node
      type(eval_node_t), intent(in), target :: arg1
      type(string_t), intent(in) :: name
      procedure(unary_cut) :: proc
      node%type = EN_LOG_FUN_UNARY
      node%tag = name
      node%result_type = V_LOG
      call eval_node_allocate_value (node)
      node%arg1 => arg1
      allocate (node%index)
      allocate (node%prt1)
      node%op1_cut => proc
    end subroutine eval_node_init_log_fun_unary

    subroutine eval_node_init_log_fun_binary (node, arg1, arg2, name, proc)
      type(eval_node_t), intent(out) :: node
      type(eval_node_t), intent(in), target :: arg1, arg2
```

```
      type(string_t), intent(in) :: name
      procedure(binary_cut) :: proc
      node%type = EN_LOG_FUN_BINARY
      node%tag = name
      node%result_type = V_LOG
      call eval_node_allocate_value (node)
      node%arg1 => arg1
      node%arg2 => arg2
      allocate (node%index)
      allocate (node%prt1)
      allocate (node%prt2)
      node%op2_cut => proc
    end subroutine eval_node_init_log_fun_binary
```

These are for particle-list functions that evaluate to an integer value.

⟨Expressions: procedures⟩+≡
```
    subroutine eval_node_init_int_fun_unary (node, arg1, name, proc)
      type(eval_node_t), intent(out) :: node
      type(eval_node_t), intent(in), target :: arg1
      type(string_t), intent(in) :: name
      procedure(unary_num) :: proc
      node%type = EN_INT_FUN_UNARY
      node%tag = name
      node%result_type = V_INT
      call eval_node_allocate_value (node)
      node%arg1 => arg1
      allocate (node%index)
      allocate (node%prt1)
      node%op1_num => proc
    end subroutine eval_node_init_int_fun_unary

    subroutine eval_node_init_int_fun_binary (node, arg1, arg2, name, proc)
      type(eval_node_t), intent(out) :: node
      type(eval_node_t), intent(in), target :: arg1, arg2
      type(string_t), intent(in) :: name
      procedure(binary_num) :: proc
      node%type = EN_INT_FUN_BINARY
      node%tag = name
      node%result_type = V_INT
      call eval_node_allocate_value (node)
      node%arg1 => arg1
      node%arg2 => arg2
      allocate (node%index)
      allocate (node%prt1)
      allocate (node%prt2)
      node%op2_num => proc
    end subroutine eval_node_init_int_fun_binary
```

Initialize a node for a string formatting function (sprintf).

⟨Expressions: procedures⟩+≡
```
    subroutine eval_node_init_format_string (node, fmt, arg, name, n_args)
      type(eval_node_t), intent(out) :: node
      type(eval_node_t), pointer :: fmt, arg
```

```
      type(string_t), intent(in) :: name
      integer, intent(in) :: n_args
      node%type = EN_FORMAT_STR
      node%tag = name
      node%result_type = V_STR
      call eval_node_allocate_value (node)
      node%arg0 => fmt
      node%arg1 => arg
      allocate (node%ival)
      node%ival = n_args
    end subroutine eval_node_init_format_string
```

If particle functions depend upon a condition (or an expression is evaluated), the observables that can be evaluated for the given particles have to be thrown on the local variable stack. This is done here. Each observable is initialized with the particle pointers which have been allocated for the node.

The integer variable that is referred to by the Index pseudo-observable is always known when it is referred to.

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_node_set_observables (node, var_list)
      type(eval_node_t), intent(inout) :: node
      type(var_list_t), intent(in), target :: var_list
      logical, save, target :: known = .true.
      allocate (node%var_list)
      call var_list_link (node%var_list, var_list)
      allocate (node%index)
      call var_list_append_int_ptr &
           (node%var_list, var_str ("Index"), node%index, known, intrinsic=.true.)
      if (.not. associated (node%prt2)) then
         call var_list_set_observables_unary &
              (node%var_list, node%prt1)
      else
         call var_list_set_observables_binary &
              (node%var_list, node%prt1, node%prt2)
      end if
    end subroutine eval_node_set_observables
```

Output

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_node_write (node, unit, indent)
      type(eval_node_t), intent(in) :: node
      integer, intent(in), optional :: unit
      integer, intent(in), optional :: indent
      integer :: u, ind
      u = output_unit (unit);  if (u < 0)  return
      ind = 0;  if (present (indent)) ind = indent
      write (u, "(A)", advance="no")  repeat ("|  ", ind) // "o "
      select case (node%type)
      case (EN_UNARY, EN_BINARY, EN_CONDITIONAL, &
            EN_PRT_FUN_UNARY, EN_PRT_FUN_BINARY, &
            EN_EVAL_FUN_UNARY, EN_EVAL_FUN_BINARY, &
```

```
           EN_LOG_FUN_UNARY, EN_LOG_FUN_BINARY, &
           EN_INT_FUN_UNARY, EN_INT_FUN_BINARY)
      write (u, "(A)", advance="no")  "[" // char (node%tag) // "] ="
case (EN_CONSTANT)
      write (u, "(A)", advance="no")  "[const] ="
case (EN_VARIABLE)
      write (u, "(A)", advance="no")  char (node%tag) // " =>"
case (EN_OBS1_INT, EN_OBS2_INT, EN_OBS1_REAL, EN_OBS2_REAL)
      write (u, "(A)", advance="no")  char (node%tag) // " ="
case (EN_BLOCK)
      write (u, "(A)", advance="no")  "[" // char (node%tag) // "]" // &
           char (node%var_name) // " [expr] = "
case default
      write (u, "(A)", advance="no")  "[???] ="
end select
select case (node%result_type)
case (V_LOG)
      if (node%value_is_known) then
         if (node%lval) then
            write (u, *) "true"
         else
            write (u, *) "false"
         end if
      else
         write (u, *) "[unknown logical]"
      end if
case (V_INT)
      if (node%value_is_known) then
         write (u, *)  node%ival
      else
         write (u, *) "[unknown integer]"
      end if
case (V_REAL)
      if (node%value_is_known) then
         write (u, *) node%rval
      else
         write (u, *) "[unknown real]"
      end if
case (V_CMPLX)
      if (node%value_is_known) then
         write (u, *) node%cval
      else
         write (u, *) "[unknown complex]"
      end if
case (V_PTL)
      if (char (node%tag) == "@evt") then
         write (u, *) "[event particle list]"
      else if (node%value_is_known) then
         call prt_list_write &
              (node%pval, unit, prefix = repeat ("|  ", ind + 1))
      else
         write (u, *) "[unknown particle list]"
      end if
case (V_PDG)
```

```
         call pdg_array_write (node%aval, u);  write (u, *)
      case (V_STR)
         if (node%value_is_known) then
            write (u, "(A)")  ’"’ // char (node%sval) // ’"’
         else
            write (u, *) "[unknown string]"
         end if
      case default
         write (u, *) "[empty]"
      end select
      select case (node%type)
      case (EN_OBS1_INT, EN_OBS1_REAL)
         write (u, "(A,6x,A)", advance="no")  repeat ("|  ", ind), "prt1 ="
         call prt_write (node%prt1, unit)
      case (EN_OBS2_INT, EN_OBS2_REAL)
         write (u, "(A,6x,A)", advance="no")  repeat ("|  ", ind), "prt1 ="
         call prt_write (node%prt1, unit)
         write (u, "(A,6x,A)", advance="no")  repeat ("|  ", ind), "prt2 ="
         call prt_write (node%prt2, unit)
      end select
    end subroutine eval_node_write

    recursive subroutine eval_node_write_rec (node, unit, indent)
      type(eval_node_t), intent(in) :: node
      integer, intent(in), optional :: unit
      integer, intent(in), optional :: indent
      integer :: u, ind
      u = output_unit (unit);  if (u < 0)  return
      ind = 0;  if (present (indent))  ind = indent
      call eval_node_write (node, unit, indent)
      select case (node%type)
      case (EN_UNARY)
         if (associated (node%arg0)) &
              call eval_node_write_rec (node%arg0, unit, ind+1)
         call eval_node_write_rec (node%arg1, unit, ind+1)
      case (EN_BINARY)
         if (associated (node%arg0)) &
              call eval_node_write_rec (node%arg0, unit, ind+1)
         call eval_node_write_rec (node%arg1, unit, ind+1)
         call eval_node_write_rec (node%arg2, unit, ind+1)
      case (EN_BLOCK)
         call eval_node_write_rec (node%arg1, unit, ind+1)
         call eval_node_write_rec (node%arg0, unit, ind+1)
      case (EN_CONDITIONAL)
         call eval_node_write_rec (node%arg0, unit, ind+1)
         call eval_node_write_rec (node%arg1, unit, ind+1)
         call eval_node_write_rec (node%arg2, unit, ind+1)
      case (EN_PRT_FUN_UNARY, EN_EVAL_FUN_UNARY, &
            EN_LOG_FUN_UNARY, EN_INT_FUN_UNARY)
         if (associated (node%arg0)) &
              call eval_node_write_rec (node%arg0, unit, ind+1)
         call eval_node_write_rec (node%arg1, unit, ind+1)
      case (EN_PRT_FUN_BINARY, EN_EVAL_FUN_BINARY, &
            EN_LOG_FUN_BINARY, EN_INT_FUN_BINARY)
```

```
            if (associated (node%arg0)) &
                 call eval_node_write_rec (node%arg0, unit, ind+1)
            call eval_node_write_rec (node%arg1, unit, ind+1)
            call eval_node_write_rec (node%arg2, unit, ind+1)
        end select
    end subroutine eval_node_write_rec
```

5.5.2   Operation types

For the operations associated to evaluation tree nodes, we define abstract interfaces for all cases.

Particles/particle lists are transferred by-reference, to avoid unnecessary copying. Therefore, subroutines instead of functions. (Furthermore, the function version of `unary_prt` triggers an obscure bug in nagfor 5.2(649) [invalid C code].)

⟨*Expressions: interfaces*⟩≡
```
    abstract interface
       logical function unary_log (arg)
         import eval_node_t
         type(eval_node_t), intent(in) :: arg
       end function unary_log
    end interface
    abstract interface
       integer function unary_int (arg)
         import eval_node_t
         type(eval_node_t), intent(in) :: arg
       end function unary_int
    end interface
    abstract interface
       real(default) function unary_real (arg)
         import default
         import eval_node_t
         type(eval_node_t), intent(in) :: arg
       end function unary_real
    end interface
    abstract interface
       complex(default) function unary_cmplx (arg)
         import default
         import eval_node_t
         type(eval_node_t), intent(in) :: arg
       end function unary_cmplx
    end interface
    abstract interface
       subroutine unary_pdg (pdg_array, arg)
         import pdg_array_t
         import eval_node_t
         type(pdg_array_t), intent(out) :: pdg_array
         type(eval_node_t), intent(in) :: arg
       end subroutine unary_pdg
    end interface
    abstract interface
       subroutine unary_ptl (prt_list, arg, arg0)
```

```fortran
      import prt_list_t
      import eval_node_t
      type(prt_list_t), intent(inout) :: prt_list
      type(eval_node_t), intent(in) :: arg
      type(eval_node_t), intent(inout), optional :: arg0
   end subroutine unary_ptl
end interface
abstract interface
   subroutine unary_str (string, arg)
     import string_t
     import eval_node_t
     type(string_t), intent(out) :: string
     type(eval_node_t), intent(in) :: arg
   end subroutine unary_str
end interface
abstract interface
   logical function unary_cut (arg1, arg0)
     import eval_node_t
     type(eval_node_t), intent(in) :: arg1
     type(eval_node_t), intent(inout) :: arg0
   end function unary_cut
end interface
abstract interface
   subroutine unary_num (ival, arg1, arg0)
     import eval_node_t
     integer, intent(out) :: ival
     type(eval_node_t), intent(in) :: arg1
     type(eval_node_t), intent(inout), optional :: arg0
   end subroutine unary_num
end interface
abstract interface
   logical function binary_log (arg1, arg2)
     import eval_node_t
     type(eval_node_t), intent(in) :: arg1, arg2
   end function binary_log
end interface
abstract interface
   integer function binary_int (arg1, arg2)
     import eval_node_t
     type(eval_node_t), intent(in) :: arg1, arg2
   end function binary_int
end interface
abstract interface
   real(default) function binary_real (arg1, arg2)
     import default
     import eval_node_t
     type(eval_node_t), intent(in) :: arg1, arg2
   end function binary_real
end interface
  abstract interface
   complex(default) function binary_cmplx (arg1, arg2)
     import default
     import eval_node_t
     type(eval_node_t), intent(in) :: arg1, arg2
```

```
        end function binary_cmplx
    end interface
    abstract interface
        subroutine binary_pdg (pdg_array, arg1, arg2)
          import pdg_array_t
          import eval_node_t
          type(pdg_array_t), intent(out) :: pdg_array
          type(eval_node_t), intent(in) :: arg1, arg2
        end subroutine binary_pdg
    end interface
    abstract interface
        subroutine binary_ptl (prt_list, arg1, arg2, arg0)
          import prt_list_t
          import eval_node_t
          type(prt_list_t), intent(inout) :: prt_list
          type(eval_node_t), intent(in) :: arg1, arg2
          type(eval_node_t), intent(inout), optional :: arg0
        end subroutine binary_ptl
    end interface
    abstract interface
        subroutine binary_str (string, arg1, arg2)
          import string_t
          import eval_node_t
          type(string_t), intent(out) :: string
          type(eval_node_t), intent(in) :: arg1, arg2
        end subroutine binary_str
    end interface
    abstract interface
        logical function binary_cut (arg1, arg2, arg0)
          import eval_node_t
          type(eval_node_t), intent(in) :: arg1, arg2
          type(eval_node_t), intent(inout) :: arg0
        end function binary_cut
    end interface
    abstract interface
        subroutine binary_num (ival, arg1, arg2, arg0)
          import eval_node_t
          integer, intent(out) :: ival
          type(eval_node_t), intent(in) :: arg1, arg2
          type(eval_node_t), intent(inout), optional :: arg0
        end subroutine binary_num
    end interface
```

The following subroutines set the procedure pointer:

⟨*Expressions: procedures*⟩+≡

```
    subroutine eval_node_set_op1_log (en, op)
      type(eval_node_t), intent(inout) :: en
      procedure(unary_log) :: op
      en%op1_log => op
      end subroutine eval_node_set_op1_log

    subroutine eval_node_set_op1_int (en, op)
      type(eval_node_t), intent(inout) :: en
      procedure(unary_int) :: op
```

```
    en%op1_int => op
end subroutine eval_node_set_op1_int

subroutine eval_node_set_op1_real (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(unary_real) :: op
  en%op1_real => op
end subroutine eval_node_set_op1_real

subroutine eval_node_set_op1_cmplx (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(unary_cmplx) :: op
  en%op1_cmplx => op
end subroutine eval_node_set_op1_cmplx

subroutine eval_node_set_op1_pdg (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(unary_pdg) :: op
  en%op1_pdg => op
end subroutine eval_node_set_op1_pdg

subroutine eval_node_set_op1_ptl (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(unary_ptl) :: op
  en%op1_ptl => op
end subroutine eval_node_set_op1_ptl

subroutine eval_node_set_op1_str (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(unary_str) :: op
  en%op1_str => op
end subroutine eval_node_set_op1_str

subroutine eval_node_set_op2_log (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_log) :: op
  en%op2_log => op
end subroutine eval_node_set_op2_log

subroutine eval_node_set_op2_int (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_int) :: op
  en%op2_int => op
end subroutine eval_node_set_op2_int

subroutine eval_node_set_op2_real (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_real) :: op
  en%op2_real => op
end subroutine eval_node_set_op2_real

  subroutine eval_node_set_op2_cmplx (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_cmplx) :: op
```

```
    en%op2_cmplx => op
end subroutine eval_node_set_op2_cmplx

subroutine eval_node_set_op2_pdg (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_pdg) :: op
  en%op2_pdg => op
end subroutine eval_node_set_op2_pdg

subroutine eval_node_set_op2_ptl (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_ptl) :: op
  en%op2_ptl => op
end subroutine eval_node_set_op2_ptl

subroutine eval_node_set_op2_str (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_str) :: op
  en%op2_str => op
end subroutine eval_node_set_op2_str
```

5.5.3 Specific operators

Our expression syntax contains all Fortran functions that make sense. These
functions have to be provided in a form that they can be used in procedures
pointers, and have the abstract interfaces above. For some intrinsic functions,
we could use specific versions provided by Fortran directly. However, this has
two drawbacks: (i) We should work with the values instead of the eval-nodes as
argument, which complicates the interface; (ii) more importantly, the `default`
real type need not be equivalent to double precision. This would, at least, intro-
duce system dependencies. Finally, for operators there are no specific versions.

   Therefore, we write wrappers for all possible functions, at the expense of
some overhead.

Binary numerical functions

⟨*Expressions: procedures*⟩+≡
```
    integer function add_ii (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival + en2%ival
    end function add_ii
    real(default) function add_ir (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival + en2%rval
    end function add_ir
    complex(default) function add_ic (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival + en2%cval
    end function add_ic
    real(default) function add_ri (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval + en2%ival
```

```
end function add_ri
complex(default) function add_ci (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval + en2%ival
end function add_ci
complex(default) function add_cr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval + en2%rval
end function add_cr
complex(default) function add_rc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval + en2%cval
end function add_rc
real(default) function add_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval + en2%rval
end function add_rr
complex(default) function add_cc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval + en2%cval
end function add_cc

integer function sub_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival - en2%ival
end function sub_ii
real(default) function sub_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival - en2%rval
end function sub_ir
real(default) function sub_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval - en2%ival
end function sub_ri
complex(default) function sub_ic (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival - en2%cval
end function sub_ic
complex(default) function sub_ci (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval - en2%ival
end function sub_ci
complex(default) function sub_cr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval - en2%rval
end function sub_cr
complex(default) function sub_rc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval - en2%cval
end function sub_rc
real(default) function sub_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval - en2%rval
end function sub_rr
```

```
complex(default) function sub_cc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval - en2%cval
end function sub_cc

integer function mul_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival * en2%ival
end function mul_ii
real(default) function mul_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival * en2%rval
end function mul_ir
real(default) function mul_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval * en2%ival
end function mul_ri
complex(default) function mul_ic (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival * en2%cval
end function mul_ic
complex(default) function mul_ci (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval * en2%ival
end function mul_ci
complex(default) function mul_rc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval * en2%cval
end function mul_rc
complex(default) function mul_cr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval * en2%rval
end function mul_cr
real(default) function mul_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval * en2%rval
end function mul_rr
complex(default) function mul_cc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval * en2%cval
end function mul_cc

integer function div_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival / en2%ival
end function div_ii
real(default) function div_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival / en2%rval
end function div_ir
real(default) function div_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval / en2%ival
end function div_ri
```

```
complex(default) function div_ic (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival / en2%cval
end function div_ic
complex(default) function div_ci (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval / en2%ival
end function div_ci
complex(default) function div_rc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval / en2%cval
end function div_rc
complex(default) function div_cr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval / en2%rval
end function div_cr
real(default) function div_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval / en2%rval
end function div_rr
complex(default) function div_cc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval / en2%cval
end function div_cc

integer function pow_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival ** en2%ival
end function pow_ii
real(default) function pow_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval ** en2%ival
end function pow_ri
complex(default) function pow_ci (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval ** en2%ival
end function pow_ci
real(default) function pow_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival ** en2%rval
end function pow_ir
real(default) function pow_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval ** en2%rval
end function pow_rr
complex(default) function pow_cr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval ** en2%rval
end function pow_cr
complex(default) function pow_ic (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival ** en2%cval
end function pow_ic
complex(default) function pow_rc (en1, en2) result (y)
```

```
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval ** en2%cval
end function pow_rc
complex(default) function pow_cc (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval ** en2%cval
end function pow_cc

integer function max_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = max (en1%ival, en2%ival)
end function max_ii
real(default) function max_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = max (real (en1%ival, default), en2%rval)
end function max_ir
real(default) function max_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = max (en1%rval, real (en2%ival, default))
end function max_ri
real(default) function max_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = max (en1%rval, en2%rval)
end function max_rr
integer function min_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = min (en1%ival, en2%ival)
end function min_ii
real(default) function min_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = min (real (en1%ival, default), en2%rval)
end function min_ir
real(default) function min_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = min (en1%rval, real (en2%ival, default))
end function min_ri
real(default) function min_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = min (en1%rval, en2%rval)
end function min_rr

integer function mod_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = mod (en1%ival, en2%ival)
end function mod_ii
real(default) function mod_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = mod (real (en1%ival, default), en2%rval)
end function mod_ir
real(default) function mod_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = mod (en1%rval, real (en2%ival, default))
end function mod_ri
real(default) function mod_rr (en1, en2) result (y)
```

```
    type(eval_node_t), intent(in) :: en1, en2
    y = mod (en1%rval, en2%rval)
  end function mod_rr
  integer function modulo_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = modulo (en1%ival, en2%ival)
  end function modulo_ii
  real(default) function modulo_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = modulo (real (en1%ival, default), en2%rval)
  end function modulo_ir
  real(default) function modulo_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = modulo (en1%rval, real (en2%ival, default))
  end function modulo_ri
  real(default) function modulo_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = modulo (en1%rval, en2%rval)
  end function modulo_rr
```

Unary numeric functions

⟨*Expressions: procedures*⟩+≡

```
  real(default) function real_i (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%ival
  end function real_i
  real(default) function real_c (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%cval
  end function real_c
  integer function int_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%rval
  end function int_r
  complex(default) function cmplx_i (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%ival
  end function cmplx_i
  integer function int_c (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%cval
  end function int_c
  complex(default) function cmplx_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%rval
  end function cmplx_r
  integer function nint_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = nint (en%rval)
  end function nint_r
  integer function floor_r (en) result (y)
    type(eval_node_t), intent(in) :: en
```

```
  y = floor (en%rval)
end function floor_r
integer function ceiling_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = ceiling (en%rval)
end function ceiling_r

integer function neg_i (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = - en%ival
end function neg_i
real(default) function neg_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = - en%rval
end function neg_r
complex(default) function neg_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = - en%cval
end function neg_c
integer function abs_i (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = abs (en%ival)
end function abs_i
real(default) function abs_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = abs (en%rval)
end function abs_r
real(default) function abs_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = abs (en%cval)
end function abs_c
integer function sgn_i (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sign (1, en%ival)
end function sgn_i
real(default) function sgn_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sign (1._default, en%rval)
end function sgn_r

real(default) function sqrt_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sqrt (en%rval)
end function sqrt_r
real(default) function exp_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = exp (en%rval)
end function exp_r
real(default) function log_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = log (en%rval)
end function log_r
real(default) function log10_r (en) result (y)
  type(eval_node_t), intent(in) :: en
```

```
    y = log10 (en%rval)
  end function log10_r

complex(default) function sqrt_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sqrt (en%cval)
end function sqrt_c
complex(default) function exp_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = exp (en%cval)
end function exp_c
complex(default) function log_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = log (en%cval)
end function log_c

real(default) function sin_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sin (en%rval)
end function sin_r
real(default) function cos_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = cos (en%rval)
end function cos_r
real(default) function tan_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = tan (en%rval)
end function tan_r
real(default) function asin_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = asin (en%rval)
end function asin_r
real(default) function acos_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = acos (en%rval)
end function acos_r
real(default) function atan_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = atan (en%rval)
end function atan_r

complex(default) function sin_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sin (en%cval)
end function sin_c
complex(default) function cos_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = cos (en%cval)
end function cos_c

real(default) function sinh_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sinh (en%rval)
end function sinh_r
```

```
    real(default) function cosh_r (en) result (y)
      type(eval_node_t), intent(in) :: en
      y = cosh (en%rval)
    end function cosh_r
    real(default) function tanh_r (en) result (y)
      type(eval_node_t), intent(in) :: en
      y = tanh (en%rval)
    end function tanh_r
!   real(default) function asinh_r (en) result (y)
!     type(eval_node_t), intent(in) :: en
!     y = asinh (en%rval)
!   end function asinh_r
!   real(default) function acosh_r (en) result (y)
!     type(eval_node_t), intent(in) :: en
!     y = acosh (en%rval)
!   end function acosh_r
!   real(default) function atanh_r (en) result (y)
!     type(eval_node_t), intent(in) :: en
!     y = atanh (en%rval)
!   end function atanh_r
```

Binary logical functions

Logical expressions:

⟨*Expressions: procedures*⟩+≡
```
    logical function or_ll (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%lval .or. en2%lval
    end function or_ll
    logical function and_ll (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%lval .and. en2%lval
    end function and_ll
```

Comparisons:

⟨*Expressions: procedures*⟩+≡
```
    logical function comp_lt_ii (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival < en2%ival
    end function comp_lt_ii
    logical function comp_lt_ir (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival < en2%rval
    end function comp_lt_ir
    logical function comp_lt_ri (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval < en2%ival
    end function comp_lt_ri
    logical function comp_lt_rr (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval < en2%rval
    end function comp_lt_rr
```

```
logical function comp_gt_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival > en2%ival
end function comp_gt_ii
logical function comp_gt_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival > en2%rval
end function comp_gt_ir
logical function comp_gt_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval > en2%ival
end function comp_gt_ri
logical function comp_gt_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval > en2%rval
end function comp_gt_rr

logical function comp_le_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival <= en2%ival
end function comp_le_ii
logical function comp_le_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival <= en2%rval
end function comp_le_ir
logical function comp_le_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval <= en2%ival
end function comp_le_ri
logical function comp_le_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval <= en2%rval
end function comp_le_rr

logical function comp_ge_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival >= en2%ival
end function comp_ge_ii
logical function comp_ge_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival >= en2%rval
end function comp_ge_ir
logical function comp_ge_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval >= en2%ival
end function comp_ge_ri
logical function comp_ge_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval >= en2%rval
end function comp_ge_rr

logical function comp_eq_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
```

```
      y = en1%ival == en2%ival
    end function comp_eq_ii
    logical function comp_eq_ir (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival == en2%rval
    end function comp_eq_ir
    logical function comp_eq_ri (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval == en2%ival
    end function comp_eq_ri
    logical function comp_eq_rr (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval == en2%rval
    end function comp_eq_rr
    logical function comp_eq_ss (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%sval == en2%sval
    end function comp_eq_ss

    logical function comp_ne_ii (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival /= en2%ival
    end function comp_ne_ii
    logical function comp_ne_ir (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival /= en2%rval
    end function comp_ne_ir
    logical function comp_ne_ri (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval /= en2%ival
    end function comp_ne_ri
    logical function comp_ne_rr (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval /= en2%rval
    end function comp_ne_rr
    logical function comp_ne_ss (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%sval /= en2%sval
    end function comp_ne_ss

    logical function comp_sim_ii (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      if (associated (en1%tolerance)) then
         y = abs (en1%ival - en2%ival) <= en1%tolerance
      else
         y = en1%ival == en2%ival
      end if
    end function comp_sim_ii
    logical function comp_sim_ri (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      if (associated (en1%tolerance)) then
         y = abs (en1%rval - en2%ival) <= en1%tolerance
      else
         y = en1%rval == en2%ival
```

```fortran
    end if
end function comp_sim_ri
logical function comp_sim_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = abs (en1%ival - en2%rval) <= en1%tolerance
  else
     y = en1%ival == en2%rval
  end if
end function comp_sim_ir
logical function comp_sim_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = abs (en1%rval - en2%rval) <= en1%tolerance
  else
     y = en1%rval == en2%rval
  end if
end function comp_sim_rr
logical function comp_nsim_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = abs (en1%ival - en2%ival) > en1%tolerance
  else
     y = en1%ival /= en2%ival
  end if
end function comp_nsim_ii
logical function comp_nsim_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = abs (en1%rval - en2%ival) > en1%tolerance
  else
     y = en1%rval /= en2%ival
  end if
end function comp_nsim_ri
logical function comp_nsim_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = abs (en1%ival - en2%rval) > en1%tolerance
  else
     y = en1%ival /= en2%rval
  end if
end function comp_nsim_ir
logical function comp_nsim_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = abs (en1%rval - en2%rval) > en1%tolerance
  else
     y = en1%rval /= en2%rval
  end if
end function comp_nsim_rr
```

Unary logical functions

⟨Expressions: procedures⟩+≡
```
logical function not_l (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = .not. en%lval
end function not_l
```

Unary PDG-array functions

Make a PDG-array object from an integer.

⟨Expressions: procedures⟩+≡
```
subroutine pdg_i (pdg_array, en)
  type(pdg_array_t), intent(out) :: pdg_array
  type(eval_node_t), intent(in) :: en
  pdg_array = en%ival
end subroutine pdg_i
```

Binary PDG-array functions

Concatenate two PDG-array objects.

⟨Expressions: procedures⟩+≡
```
subroutine concat_cc (pdg_array, en1, en2)
  type(pdg_array_t), intent(out) :: pdg_array
  type(eval_node_t), intent(in) :: en1, en2
  pdg_array = en1%aval // en2%aval
end subroutine concat_cc
```

Unary particle-list functions

Combine all particles of the first argument. If en0 is present, create a mask which is true only for those particles that pass the test.

⟨Expressions: procedures⟩+≡
```
subroutine collect_p (prt_list, en1, en0)
  type(prt_list_t), intent(inout) :: prt_list
  type(eval_node_t), intent(in) :: en1
  type(eval_node_t), intent(inout), optional :: en0
  logical, dimension(:), allocatable :: mask1
  integer :: n, i
  n = prt_list_get_length (en1%pval)
  allocate (mask1 (n))
  if (present (en0)) then
     do i = 1, n
        en0%index = i
        en0%prt1 = prt_list_get_prt (en1%pval, i)
        call eval_node_evaluate (en0)
        mask1(i) = en0%lval
     end do
  else
     mask1 = .true.
```

```
      end if
      call prt_list_collect (prt_list, en1%pval, mask1)
    end subroutine collect_p
```

Select all particles of the first argument. If en0 is present, create a mask which is true only for those particles that pass the test.

⟨*Expressions: procedures*⟩+≡
```
    subroutine select_p (prt_list, en1, en0)
      type(prt_list_t), intent(inout) :: prt_list
      type(eval_node_t), intent(in) :: en1
      type(eval_node_t), intent(inout), optional :: en0
      logical, dimension(:), allocatable :: mask1
      integer :: n, i
      n = prt_list_get_length (en1%pval)
      allocate (mask1 (n))
      if (present (en0)) then
         do i = 1, prt_list_get_length (en1%pval)
            en0%index = i
            en0%prt1 = prt_list_get_prt (en1%pval, i)
            call eval_node_evaluate (en0)
            mask1(i) = en0%lval
         end do
      else
         mask1 = .true.
      end if
      call prt_list_select (prt_list, en1%pval, mask1)
    end subroutine select_p
```

Extract the particle with index given by en0 from the argument list. Negative indices count from the end. If en0 is absent, extract the first particle. The result is a list with a single entry, or no entries if the original list was empty or if the index is out of range.

This function has no counterpart with two arguments.

⟨*Expressions: procedures*⟩+≡
```
    subroutine extract_p (prt_list, en1, en0)
      type(prt_list_t), intent(inout) :: prt_list
      type(eval_node_t), intent(in) :: en1
      type(eval_node_t), intent(inout), optional :: en0
      integer :: index
      if (present (en0)) then
         call eval_node_evaluate (en0)
         select case (en0%result_type)
         case (V_INT);  index = en0%ival
         case default
            call eval_node_write (en0)
            call msg_fatal (" Index parameter of 'extract' must be integer.")
         end select
      else
         index = 1
      end if
      call prt_list_extract (prt_list, en1%pval, index)
    end subroutine extract_p
```

Sort the particle list according to the result of evaluating en0. If en0 is absent, sort by default method (PDG code, particles before antiparticles).

⟨*Expressions: procedures*⟩+≡

```
subroutine sort_p (prt_list, en1, en0)
  type(prt_list_t), intent(inout) :: prt_list
  type(eval_node_t), intent(in) :: en1
  type(eval_node_t), intent(inout), optional :: en0
  integer, dimension(:), allocatable :: ival
  real(default), dimension(:), allocatable :: rval
  integer :: i, n
  n = prt_list_get_length (en1%pval)
  if (present (en0)) then
     select case (en0%result_type)
     case (V_INT);  allocate (ival (n))
     case (V_REAL); allocate (rval (n))
     end select
     do i = 1, n
        en0%index = i
        en0%prt1 = prt_list_get_prt (en1%pval, i)
        call eval_node_evaluate (en0)
        select case (en0%result_type)
        case (V_INT);  ival(i) = en0%ival
        case (V_REAL); rval(i) = en0%rval
        end select
     end do
     select case (en0%result_type)
     case (V_INT);  call prt_list_sort (prt_list, en1%pval, ival)
     case (V_REAL); call prt_list_sort (prt_list, en1%pval, rval)
     end select
  else
     call prt_list_sort (prt_list, en1%pval)
  end if
end subroutine sort_p
```

The following functions return a logical value. `all` evaluates to true if the condition en0 is true for all elements of the particle list. `any` and `no` are analogous.

⟨*Expressions: procedures*⟩+≡

```
function all_p (en1, en0) result (lval)
  logical :: lval
  type(eval_node_t), intent(in) :: en1
  type(eval_node_t), intent(inout) :: en0
  integer :: i, n
  n = prt_list_get_length (en1%pval)
  lval = .true.
  do i = 1, n
     en0%index = i
     en0%prt1 = prt_list_get_prt (en1%pval, i)
     call eval_node_evaluate (en0)
     lval = en0%lval
     if (.not. lval)  exit
  end do
end function all_p
```

```
function any_p (en1, en0) result (lval)
  logical :: lval
  type(eval_node_t), intent(in) :: en1
  type(eval_node_t), intent(inout) :: en0
  integer :: i, n
  n = prt_list_get_length (en1%pval)
  lval = .false.
  do i = 1, n
     en0%index = i
     en0%prt1 = prt_list_get_prt (en1%pval, i)
     call eval_node_evaluate (en0)
     lval = en0%lval
     if (lval)  exit
  end do
end function any_p

function no_p (en1, en0) result (lval)
  logical :: lval
  type(eval_node_t), intent(in) :: en1
  type(eval_node_t), intent(inout) :: en0
  integer :: i, n
  n = prt_list_get_length (en1%pval)
  lval = .true.
  do i = 1, n
     en0%index = i
     en0%prt1 = prt_list_get_prt (en1%pval, i)
     call eval_node_evaluate (en0)
     lval = .not. en0%lval
     if (lval)  exit
  end do
end function no_p
```

The following function returns an integer value, namely the number of particles
for which the condition is true. If there is no condition, it returns simply the
length of the particle list.

A function would be more natural. Making it a subroutine avoids another
compiler bug (internal error in nagfor 5.2 (649)). (See the interface `unary_num`.)

⟨*Expressions: procedures*⟩+≡
```
subroutine count_a (ival, en1, en0)
  integer, intent(out) :: ival
  type(eval_node_t), intent(in) :: en1
  type(eval_node_t), intent(inout), optional :: en0
  integer :: i, n, count
  n = prt_list_get_length (en1%pval)
  if (present (en0)) then
     count = 0
     do i = 1, n
        en0%index = i
        en0%prt1 = prt_list_get_prt (en1%pval, i)
        call eval_node_evaluate (en0)
        if (en0%lval)  count = count + 1
     end do
     ival = count
```

```
        else
            ival = n
        end if
    end subroutine count_a
```

Binary particle-list functions

This joins two particle lists, stored in the evaluation nodes en1 and en2. If en0 is also present, it amounts to a logical test returning true or false for every pair of particles. A particle of the second list gets a mask entry only if it passes the test for all particles of the first list.

⟨*Expressions: procedures*⟩+≡
```
    subroutine join_pp (prt_list, en1, en2, en0)
      type(prt_list_t), intent(inout) :: prt_list
      type(eval_node_t), intent(in) :: en1, en2
      type(eval_node_t), intent(inout), optional :: en0
      logical, dimension(:), allocatable :: mask2
      integer :: i, j, n1, n2
      n1 = prt_list_get_length (en1%pval)
      n2 = prt_list_get_length (en2%pval)
      allocate (mask2 (n2))
      mask2 = .true.
      if (present (en0)) then
         do i = 1, n1
            en0%index = i
            en0%prt1 = prt_list_get_prt (en1%pval, i)
            do j = 1, n2
               en0%prt2 = prt_list_get_prt (en2%pval, j)
               call eval_node_evaluate (en0)
               mask2(j) = mask2(j) .and. en0%lval
            end do
         end do
      end if
      call prt_list_join (prt_list, en1%pval, en2%pval, mask2)
    end subroutine join_pp
```

Combine two particle lists, i.e., make a list of composite particles built from all possible particle pairs from the two lists. If en0 is present, create a mask which is true only for those pairs that pass the test.

⟨*Expressions: procedures*⟩+≡
```
    subroutine combine_pp (prt_list, en1, en2, en0)
      type(prt_list_t), intent(inout) :: prt_list
      type(eval_node_t), intent(in) :: en1, en2
      type(eval_node_t), intent(inout), optional :: en0
      logical, dimension(:,:), allocatable :: mask12
      integer :: i, j, n1, n2
      n1 = prt_list_get_length (en1%pval)
      n2 = prt_list_get_length (en2%pval)
      if (present (en0)) then
         allocate (mask12 (n1, n2))
         do i = 1, n1
            en0%index = i
```

```
                en0%prt1 = prt_list_get_prt (en1%pval, i)
                do j = 1, n2
                   en0%prt2 = prt_list_get_prt (en2%pval, j)
                   call eval_node_evaluate (en0)
                   mask12(i,j) = en0%lval
                end do
             end do
             call prt_list_combine (prt_list, en1%pval, en2%pval, mask12)
          else
             call prt_list_combine (prt_list, en1%pval, en2%pval)
          end if
       end subroutine combine_pp
```

Combine all particles of the first argument. If `en0` is present, create a mask
which is true only for those particles that pass the test w.r.t. all particles in the
second argument. If `en0` is absent, the second argument is ignored.

⟨*Expressions: procedures*⟩+≡
```
       subroutine collect_pp (prt_list, en1, en2, en0)
         type(prt_list_t), intent(inout) :: prt_list
         type(eval_node_t), intent(in) :: en1, en2
         type(eval_node_t), intent(inout), optional :: en0
         logical, dimension(:), allocatable :: mask1
         integer :: i, j, n1, n2
         n1 = prt_list_get_length (en1%pval)
         n2 = prt_list_get_length (en2%pval)
         allocate (mask1 (n1))
         mask1 = .true.
         if (present (en0)) then
            do i = 1, n1
               en0%index = i
               en0%prt1 = prt_list_get_prt (en1%pval, i)
               do j = 1, n2
                  en0%prt2 = prt_list_get_prt (en2%pval, j)
                  call eval_node_evaluate (en0)
                  mask1(i) = mask1(i) .and. en0%lval
               end do
            end do
         end if
         call prt_list_collect (prt_list, en1%pval, mask1)
       end subroutine collect_pp
```

Select all particles of the first argument. If `en0` is present, create a mask which
is true only for those particles that pass the test w.r.t. all particles in the
second argument. If `en0` is absent, the second argument is ignored, and the first
argument is transferred unchanged. (This case is not very useful, of course.)

⟨*Expressions: procedures*⟩+≡
```
       subroutine select_pp (prt_list, en1, en2, en0)
         type(prt_list_t), intent(inout) :: prt_list
         type(eval_node_t), intent(in) :: en1, en2
         type(eval_node_t), intent(inout), optional :: en0
         logical, dimension(:), allocatable :: mask1
         integer :: i, j, n1, n2
         n1 = prt_list_get_length (en1%pval)
```

```
        n2 = prt_list_get_length (en2%pval)
        allocate (mask1 (n1))
        mask1 = .true.
        if (present (en0)) then
            do i = 1, n1
                en0%index = i
                en0%prt1 = prt_list_get_prt (en1%pval, i)
                do j = 1, n2
                    en0%prt2 = prt_list_get_prt (en2%pval, j)
                    call eval_node_evaluate (en0)
                    mask1(i) = mask1(i) .and. en0%lval
                end do
            end do
        end if
        call prt_list_select (prt_list, en1%pval, mask1)
    end subroutine select_pp
```

Sort the first particle list according to the result of evaluating en0. From the second particle list, only the first element is taken as reference. If en0 is absent, we sort by default method (PDG code, particles before antiparticles).

⟨*Expressions: procedures*⟩+≡

```
    subroutine sort_pp (prt_list, en1, en2, en0)
      type(prt_list_t), intent(inout) :: prt_list
      type(eval_node_t), intent(in) :: en1, en2
      type(eval_node_t), intent(inout), optional :: en0
      integer, dimension(:), allocatable :: ival
      real(default), dimension(:), allocatable :: rval
      integer :: i, n1, n2
      n1 = prt_list_get_length (en1%pval)
      n2 = prt_list_get_length (en2%pval)
      if (present (en0)) then
         select case (en0%result_type)
         case (V_INT);  allocate (ival (n1))
         case (V_REAL); allocate (rval (n1))
         end select
         do i = 1, n1
            en0%index = i
            en0%prt1 = prt_list_get_prt (en1%pval, i)
            en0%prt2 = prt_list_get_prt (en2%pval, 1)
            call eval_node_evaluate (en0)
            select case (en0%result_type)
            case (V_INT);  ival(i) = en0%ival
            case (V_REAL); rval(i) = en0%rval
            end select
         end do
         select case (en0%result_type)
         case (V_INT);  call prt_list_sort (prt_list, en1%pval, ival)
         case (V_REAL); call prt_list_sort (prt_list, en1%pval, rval)
         end select
      else
         call prt_list_sort (prt_list, en1%pval)
      end if
    end subroutine sort_pp
```

The following functions return a logical value. `all` evaluates to true if the condition `en0` is true for all valid element pairs of both particle lists. Invalid pairs (with common `src` entry) are ignored.

`any` and `no` are analogous.

⟨Expressions: procedures⟩+≡

```
function all_pp (en1, en2, en0) result (lval)
  logical :: lval
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout) :: en0
  integer :: i, j, n1, n2
  n1 = prt_list_get_length (en1%pval)
  n2 = prt_list_get_length (en2%pval)
  lval = .true.
  LOOP1: do i = 1, n1
     en0%index = i
     en0%prt1 = prt_list_get_prt (en1%pval, i)
     do j = 1, n2
        en0%prt2 = prt_list_get_prt (en2%pval, j)
        if (are_disjoint (en0%prt1, en0%prt2)) then
           call eval_node_evaluate (en0)
           lval = en0%lval
           if (.not. lval)  exit LOOP1
        end if
     end do
  end do LOOP1
end function all_pp

function any_pp (en1, en2, en0) result (lval)
  logical :: lval
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout) :: en0
  integer :: i, j, n1, n2
  n1 = prt_list_get_length (en1%pval)
  n2 = prt_list_get_length (en2%pval)
  lval = .false.
  LOOP1: do i = 1, n1
     en0%index = i
     en0%prt1 = prt_list_get_prt (en1%pval, i)
     do j = 1, n2
        en0%prt2 = prt_list_get_prt (en2%pval, j)
        if (are_disjoint (en0%prt1, en0%prt2)) then
           call eval_node_evaluate (en0)
           lval = en0%lval
           if (lval)  exit LOOP1
        end if
     end do
  end do LOOP1
end function any_pp

function no_pp (en1, en2, en0) result (lval)
  logical :: lval
  type(eval_node_t), intent(in) :: en1, en2
```

```
type(eval_node_t), intent(inout) :: en0
integer :: i, j, n1, n2
n1 = prt_list_get_length (en1%pval)
n2 = prt_list_get_length (en2%pval)
lval = .true.
LOOP1: do i = 1, n1
   en0%index = i
   en0%prt1 = prt_list_get_prt (en1%pval, i)
   do j = 1, n2
      en0%prt2 = prt_list_get_prt (en2%pval, j)
      if (are_disjoint (en0%prt1, en0%prt2)) then
         call eval_node_evaluate (en0)
         lval = .not. en0%lval
         if (lval)  exit LOOP1
      end if
   end do
end do LOOP1
end function no_pp
```

The following function returns an integer value, namely the number of valid
particle-pairs from both lists for which the condition is true. Invalid pairs (with
common src entry) are ignored. If there is no condition, it returns the number
of valid particle pairs.

A function would be more natural. Making it a subroutine avoids an-
other compiler bug (internal error in nagfor 5.2 (649)). (See the interface
binary_num.)

⟨*Expressions: procedures*⟩+≡
```
subroutine count_pp (ival, en1, en2, en0)
  integer, intent(out) :: ival
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout), optional :: en0
  integer :: i, j, n1, n2, count
  n1 = prt_list_get_length (en1%pval)
  n2 = prt_list_get_length (en2%pval)
  if (present (en0)) then
     count = 0
     do i = 1, n1
        en0%index = i
        en0%prt1 = prt_list_get_prt (en1%pval, i)
        do j = 1, n2
           en0%prt2 = prt_list_get_prt (en2%pval, j)
           if (are_disjoint (en0%prt1, en0%prt2)) then
              call eval_node_evaluate (en0)
              if (en0%lval)  count = count + 1
           end if
        end do
     end do
  else
     count = 0
     do i = 1, n1
        do j = 1, n2
           if (are_disjoint (prt_list_get_prt (en1%pval, i), &
                             prt_list_get_prt (en2%pval, j))) then
```

```
                count = count + 1
             end if
          end do
       end do
    end if
    ival = count
end subroutine count_pp
```

This function makes up a particle list from the second argument which consists only of particles which match the PDG code array (first argument).

⟨*Expressions: procedures*⟩+≡
```
subroutine select_pdg_ca (prt_list, en1, en2, en0)
  type(prt_list_t), intent(inout) :: prt_list
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout), optional :: en0
  if (present (en0)) then
     call prt_list_select_pdg_code (prt_list, en1%aval, en2%pval, en0%ival)
  else
     call prt_list_select_pdg_code (prt_list, en1%aval, en2%pval)
  end if
end subroutine select_pdg_ca
```

Binary string functions

Currently, the only string operation is concatenation.

⟨*Expressions: procedures*⟩+≡
```
subroutine concat_ss (string, en1, en2)
  type(string_t), intent(out) :: string
  type(eval_node_t), intent(in) :: en1, en2
  string = en1%sval // en2%sval
end subroutine concat_ss
```

5.5.4   Compiling the parse tree

The evaluation tree is built recursively by following a parse tree. Debug option:

⟨*Expressions: variables*⟩≡
```
logical, parameter :: debug = .false.
```

Evaluate an expression. The requested type is given as an optional argument; default is numeric (integer or real).

⟨*Expressions: procedures*⟩+≡
```
recursive subroutine eval_node_compile_genexpr &
    (en, pn, var_list, result_type)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  integer, intent(in), optional :: result_type
  if (debug) then
     print *, "read genexpr";  call parse_node_write (pn)
```

```
      end if
      if (present (result_type)) then
         select case (result_type)
         case (V_INT, V_REAL, V_CMPLX)
            call eval_node_compile_expr  (en, pn, var_list)
         case (V_LOG)
            call eval_node_compile_lexpr (en, pn, var_list)
         case (V_PTL)
            call eval_node_compile_pexpr (en, pn, var_list)
         case (V_PDG)
            call eval_node_compile_cexpr (en, pn, var_list)
         case (V_STR)
            call eval_node_compile_sexpr (en, pn, var_list)
         end select
      else
         call eval_node_compile_expr  (en, pn, var_list)
      end if
      if (debug) then
         call eval_node_write (en)
         print *, "done genexpr"
      end if
   end subroutine eval_node_compile_genexpr
```

Numeric expressions

This procedure compiles a numerical expression. This is a single term or a sum
or difference of terms. We have to account for all combinations of integer and
real arguments. If both are constant, we immediately do the calculation and
allocate a constant node.

⟨*Expressions: procedures*⟩+≡

```
   recursive subroutine eval_node_compile_expr (en, pn, var_list)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     type(parse_node_t), pointer :: pn_term, pn_addition, pn_op, pn_arg
     type(eval_node_t), pointer :: en1, en2
     type(string_t) :: key
     integer :: t1, t2, t
     if (debug) then
        print *, "read expr";  call parse_node_write (pn)
     end if
     pn_term => parse_node_get_sub_ptr (pn)
     select case (char (parse_node_get_rule_key (pn_term)))
     case ("term")
        call eval_node_compile_term (en, pn_term, var_list)
        pn_addition => parse_node_get_next_ptr (pn_term, tag="addition")
     case ("addition")
        en => null ()
        pn_addition => pn_term
     case default
        call parse_node_mismatch ("term|addition", pn)
     end select
```

```
do while (associated (pn_addition))
   pn_op => parse_node_get_sub_ptr (pn_addition)
   pn_arg => parse_node_get_next_ptr (pn_op, tag="term")
   call eval_node_compile_term (en2, pn_arg, var_list)
   t2 = en2%result_type
   if (associated (en)) then
      en1 => en
      t1 = en1%result_type
   else
      allocate (en1)
      select case (t2)
      case (V_INT);  call eval_node_init_int  (en1, 0)
      case (V_REAL); call eval_node_init_real (en1, 0._default)
      case (V_CMPLX); call eval_node_init_cmplx (en1, cmplx &
                     (0._default, 0._default, kind=default))
      end select
      t1 = t2
   end if
   t = numeric_result_type (t1, t2)
   allocate (en)
   key = parse_node_get_key (pn_op)
   if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
      select case (char (key))
      case ("+")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);  call eval_node_init_int  (en, add_ii (en1, en2))
            case (V_REAL); call eval_node_init_real (en, add_ir (en1, en2))
            case (V_CMPLX); call eval_node_init_cmplx (en, add_ic (en1, en2))
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);  call eval_node_init_real (en, add_ri (en1, en2))
            case (V_REAL); call eval_node_init_real (en, add_rr (en1, en2))
            case (V_CMPLX); call eval_node_init_cmplx (en, add_rc (en1, en2))
            end select
          case (V_CMPLX)
            select case (t2)
            case (V_INT);  call eval_node_init_cmplx (en, add_ci (en1, en2))
            case (V_REAL); call eval_node_init_cmplx (en, add_cr (en1, en2))
            case (V_CMPLX); call eval_node_init_cmplx (en, add_cc (en1, en2))
            end select
         end select
      case ("-")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT); call eval_node_init_int  (en, sub_ii (en1, en2))
            case (V_REAL); call eval_node_init_real (en, sub_ir (en1, en2))
            case (V_CMPLX); call eval_node_init_cmplx (en, sub_ic (en1, en2))
            end select
         case (V_REAL)
            select case (t2)
```

```
                case (V_INT);  call eval_node_init_real (en, sub_ri (en1, en2))
                case (V_REAL); call eval_node_init_real (en, sub_rr (en1, en2))
                case (V_CMPLX); call eval_node_init_cmplx (en, sub_rc (en1, en2))
                end select
          case (V_CMPLX)
             select case (t2)
             case (V_INT);  call eval_node_init_cmplx (en, sub_ci (en1, en2))
             case (V_REAL); call eval_node_init_cmplx (en, sub_cr (en1, en2))
             case (V_CMPLX); call eval_node_init_cmplx (en, sub_cc (en1, en2))
             end select
          end select
       end select
       call eval_node_final_rec (en1)
       call eval_node_final_rec (en2)
       deallocate (en1, en2)
    else
       call eval_node_init_branch (en, key, t, en1, en2)
       select case (char (key))
       case ("+")
          select case (t1)
          case (V_INT)
             select case (t2)
             case (V_INT);  call eval_node_set_op2_int  (en, add_ii)
             case (V_REAL); call eval_node_set_op2_real (en, add_ir)
             case (V_CMPLX);  call eval_node_set_op2_cmplx (en, add_ic)
             end select
          case (V_REAL)
             select case (t2)
             case (V_INT);  call eval_node_set_op2_real (en, add_ri)
             case (V_REAL); call eval_node_set_op2_real (en, add_rr)
             case (V_CMPLX);  call eval_node_set_op2_cmplx (en, add_rc)
             end select
          case (V_CMPLX)
             select case (t2)
             case (V_INT);  call eval_node_set_op2_cmplx (en, add_ci)
             case (V_REAL); call eval_node_set_op2_cmplx (en, add_cr)
             case (V_CMPLX);  call eval_node_set_op2_cmplx (en, add_cc)
             end select
          end select
       case ("-")
          select case (t1)
          case (V_INT)
             select case (t2)
             case (V_INT);  call eval_node_set_op2_int  (en, sub_ii)
             case (V_REAL); call eval_node_set_op2_real (en, sub_ir)
             case (V_CMPLX);  call eval_node_set_op2_cmplx (en, sub_ic)
             end select
          case (V_REAL)
             select case (t2)
             case (V_INT);  call eval_node_set_op2_real (en, sub_ri)
             case (V_REAL); call eval_node_set_op2_real (en, sub_rr)
             case (V_CMPLX);  call eval_node_set_op2_cmplx (en, sub_rc)
             end select
          case (V_CMPLX)
```

```
                    select case (t2)
                    case (V_INT);   call eval_node_set_op2_cmplx (en, sub_ci)
                    case (V_REAL);  call eval_node_set_op2_cmplx (en, sub_cr)
                    case (V_CMPLX); call eval_node_set_op2_cmplx (en, sub_cc)
                    end select
                 end select
              end select
           end if
           pn_addition => parse_node_get_next_ptr (pn_addition)
        end do
        if (debug) then
           call eval_node_write (en)
           print *, "done expr"
        end if
      end subroutine eval_node_compile_expr
```

⟨*Expressions: procedures*⟩+≡

```
      recursive subroutine eval_node_compile_term (en, pn, var_list)
        type(eval_node_t), pointer :: en
        type(parse_node_t), intent(in) :: pn
        type(var_list_t), intent(in), target :: var_list
        type(parse_node_t), pointer :: pn_factor, pn_multiplication, pn_op, pn_arg
        type(eval_node_t), pointer :: en1, en2
        type(string_t) :: key
        integer :: t1, t2, t
        if (debug) then
           print *, "read term";  call parse_node_write (pn)
        end if
        pn_factor => parse_node_get_sub_ptr (pn, tag="factor")
        call eval_node_compile_factor (en, pn_factor, var_list)
        pn_multiplication => &
             parse_node_get_next_ptr (pn_factor, tag="multiplication")
        do while (associated (pn_multiplication))
           pn_op => parse_node_get_sub_ptr (pn_multiplication)
           pn_arg => parse_node_get_next_ptr (pn_op, tag="factor")
           en1 => en
           call eval_node_compile_factor (en2, pn_arg, var_list)
           t1 = en1%result_type
           t2 = en2%result_type
           t = numeric_result_type (t1, t2)
           allocate (en)
           key = parse_node_get_key (pn_op)
           if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
              select case (char (key))
              case ("*")
                 select case (t1)
                 case (V_INT)
                    select case (t2)
                    case (V_INT);   call eval_node_init_int  (en, mul_ii (en1, en2))
                    case (V_REAL);  call eval_node_init_real (en, mul_ir (en1, en2))
                    case (V_CMPLX); call eval_node_init_cmplx (en, mul_ic (en1, en2))
                    end select
                 case (V_REAL)
                    select case (t2)
```

```
                  case (V_INT);  call eval_node_init_real  (en, mul_ri (en1, en2))
                  case (V_REAL); call eval_node_init_real  (en, mul_rr (en1, en2))
                  case (V_CMPLX); call eval_node_init_cmplx (en, mul_rc (en1, en2))
                  end select
              case (V_CMPLX)
                select case (t2)
                case (V_INT);  call eval_node_init_cmplx (en, mul_ci (en1, en2))
                case (V_REAL); call eval_node_init_cmplx (en, mul_cr (en1, en2))
                case (V_CMPLX); call eval_node_init_cmplx (en, mul_cc (en1, en2))
                end select
            end select
        case ("/")
            select case (t1)
            case (V_INT)
                select case (t2)
                case (V_INT);  call eval_node_init_int  (en, div_ii (en1, en2))
                case (V_REAL); call eval_node_init_real (en, div_ir (en1, en2))
                case (V_CMPLX); call eval_node_init_real (en, div_ir (en1, en2))
                end select
            case (V_REAL)
                select case (t2)
                case (V_INT);  call eval_node_init_real (en, div_ri (en1, en2))
                case (V_REAL); call eval_node_init_real (en, div_rr (en1, en2))
                case (V_CMPLX); call eval_node_init_cmplx (en, div_rc (en1, en2))
                end select
            case (V_CMPLX)
                select case (t2)
                case (V_INT);  call eval_node_init_cmplx (en, div_ci (en1, en2))
                case (V_REAL); call eval_node_init_cmplx (en, div_cr (en1, en2))
                case (V_CMPLX); call eval_node_init_cmplx (en, div_cc (en1, en2))
                end select
            end select
        end select
        call eval_node_final_rec (en1)
        call eval_node_final_rec (en2)
        deallocate (en1, en2)
    else
        call eval_node_init_branch (en, key, t, en1, en2)
        select case (char (key))
        case ("*")
            select case (t1)
            case (V_INT)
                select case (t2)
                case (V_INT);  call eval_node_set_op2_int  (en, mul_ii)
                case (V_REAL); call eval_node_set_op2_real (en, mul_ir)
                case (V_CMPLX); call eval_node_set_op2_cmplx (en, mul_ic)
                end select
            case (V_REAL)
                select case (t2)
                case (V_INT);  call eval_node_set_op2_real (en, mul_ri)
                case (V_REAL); call eval_node_set_op2_real (en, mul_rr)
                case (V_CMPLX); call eval_node_set_op2_cmplx (en, mul_rc)
                end select
            case (V_CMPLX)
```

```
                    select case (t2)
                    case (V_INT);   call eval_node_set_op2_cmplx (en, mul_ci)
                    case (V_REAL);  call eval_node_set_op2_cmplx (en, mul_cr)
                    case (V_CMPLX); call eval_node_set_op2_cmplx (en, mul_cc)
                    end select
                 end select
             case ("/")
                select case (t1)
                case (V_INT)
                    select case (t2)
                    case (V_INT);   call eval_node_set_op2_int  (en, div_ii)
                    case (V_REAL);  call eval_node_set_op2_real (en, div_ir)
                    case (V_CMPLX); call eval_node_set_op2_cmplx (en, div_ic)
                    end select
                case (V_REAL)
                    select case (t2)
                    case (V_INT);   call eval_node_set_op2_real (en, div_ri)
                    case (V_REAL);  call eval_node_set_op2_real (en, div_rr)
                    case (V_CMPLX); call eval_node_set_op2_cmplx (en, div_rc)
                    end select
                case (V_CMPLX)
                    select case (t2)
                    case (V_INT);   call eval_node_set_op2_cmplx (en, div_ci)
                    case (V_REAL);  call eval_node_set_op2_cmplx (en, div_cr)
                    case (V_CMPLX); call eval_node_set_op2_cmplx (en, div_cc)
                    end select
                end select
             end select
          end if
          pn_multiplication => parse_node_get_next_ptr (pn_multiplication)
       end do
       if (debug) then
          call eval_node_write (en)
          print *, "done term"
       end if
    end subroutine eval_node_compile_term
```

⟨*Expressions: procedures*⟩+≡

```
    recursive subroutine eval_node_compile_factor (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_value, pn_exponentiation, pn_op, pn_arg
      type(eval_node_t), pointer :: en1, en2
      type(string_t) :: key
      integer :: t1, t2, t
      if (debug) then
         print *, "read factor";  call parse_node_write (pn)
      end if
      pn_value => parse_node_get_sub_ptr (pn)
      call eval_node_compile_signed_value (en, pn_value, var_list)
      pn_exponentiation => &
           parse_node_get_next_ptr (pn_value, tag="exponentiation")
      if (associated (pn_exponentiation)) then
```

```
pn_op => parse_node_get_sub_ptr (pn_exponentiation)
pn_arg => parse_node_get_next_ptr (pn_op)
en1 => en
call eval_node_compile_signed_value (en2, pn_arg, var_list)
t1 = en1%result_type
t2 = en2%result_type
t = numeric_result_type (t1, t2)
allocate (en)
key = parse_node_get_key (pn_op)
if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
   select case (t1)
   case (V_INT)
      select case (t2)
      case (V_INT);   call eval_node_init_int   (en, pow_ii (en1, en2))
      case (V_REAL);  call eval_node_init_real  (en, pow_ir (en1, en2))
      case (V_CMPLX); call eval_node_init_cmplx (en, pow_ic (en1, en2))
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);   call eval_node_init_real  (en, pow_ri (en1, en2))
      case (V_REAL);  call eval_node_init_real  (en, pow_rr (en1, en2))
      case (V_CMPLX); call eval_node_init_cmplx (en, pow_rc (en1, en2))
      end select
   case (V_CMPLX)
      select case (t2)
      case (V_INT);   call eval_node_init_cmplx (en, pow_ci (en1, en2))
      case (V_REAL);  call eval_node_init_cmplx (en, pow_cr (en1, en2))
      case (V_CMPLX); call eval_node_init_cmplx (en, pow_cc (en1, en2))
      end select
   end select
   call eval_node_final_rec (en1)
   call eval_node_final_rec (en2)
   deallocate (en1, en2)
else
   call eval_node_init_branch (en, key, t, en1, en2)
   select case (t1)
   case (V_INT)
      select case (t2)
      case (V_INT);   call eval_node_set_op2_int   (en, pow_ii)
      case (V_REAL,V_CMPLX);  call eval_type_error (pn, "exponentiation", t1)
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);   call eval_node_set_op2_real (en, pow_ri)
      case (V_REAL);  call eval_node_set_op2_real (en, pow_rr)
      case (V_CMPLX);  call eval_type_error (pn, "exponentiation", t1)
      end select
   case (V_CMPLX)
      select case (t2)
      case (V_INT);   call eval_node_set_op2_cmplx (en, pow_ci)
      case (V_REAL);  call eval_node_set_op2_cmplx (en, pow_cr)
      case (V_CMPLX);  call eval_node_set_op2_cmplx (en, pow_cc)
      end select
   end select
```

```
          end if
        end if
        if (debug) then
            call eval_node_write (en)
            print *, "done factor"
        end if
    end subroutine eval_node_compile_factor
```

⟨*Expressions: procedures*⟩+≡
```
    recursive subroutine eval_node_compile_signed_value (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_arg
      type(eval_node_t), pointer :: en1
      integer :: t
      if (debug) then
          print *, "read signed value";  call parse_node_write (pn)
      end if
      select case (char (parse_node_get_rule_key (pn)))
      case ("signed_value")
         pn_arg => parse_node_get_sub_ptr (pn, 2)
         call eval_node_compile_value (en1, pn_arg, var_list)
         t = en1%result_type
         allocate (en)
         if (en1%type == EN_CONSTANT) then
            select case (t)
            case (V_INT);   call eval_node_init_int   (en, neg_i (en1))
            case (V_REAL);  call eval_node_init_real  (en, neg_r (en1))
            case (V_CMPLX); call eval_node_init_cmplx (en, neg_c (en1))
            end select
            call eval_node_final_rec (en1)
            deallocate (en1)
         else
            call eval_node_init_branch (en, var_str ("-"), t, en1)
            select case (t)
            case (V_INT);   call eval_node_set_op1_int   (en, neg_i)
            case (V_REAL);  call eval_node_set_op1_real  (en, neg_r)
            case (V_CMPLX); call eval_node_set_op1_cmplx (en, neg_c)
            end select
         end if
      case default
         call eval_node_compile_value (en, pn, var_list)
      end select
      if (debug) then
          call eval_node_write (en)
          print *, "done signed value"
      end if
    end subroutine eval_node_compile_signed_value
```

Integer, real and complex values have an optional unit. The unit is extracted
and applied immediately. An integer with unit evaluates to a real constant.

⟨*Expressions: procedures*⟩+≡

```
recursive subroutine eval_node_compile_value (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  if (debug) then
     print *, "read value";  call parse_node_write (pn)
  end if
  select case (char (parse_node_get_rule_key (pn)))
  case ("integer_value", "real_value", "complex_value")
     call eval_node_compile_numeric_value (en, pn)
  case ("pi")
     call eval_node_compile_constant (en, pn)
  case ("I")
     call eval_node_compile_constant (en, pn)
  case ("variable")
     call eval_node_compile_variable (en, pn, var_list)
  case ("result")
     call eval_node_compile_result (en, pn, var_list)
  case ("expr")
     call eval_node_compile_expr (en, pn, var_list)
  case ("block_expr")
     call eval_node_compile_block_expr (en, pn, var_list)
  case ("conditional_expr")
     call eval_node_compile_conditional (en, pn, var_list)
  case ("unary_function")
     call eval_node_compile_unary_function (en, pn, var_list)
  case ("binary_function")
     call eval_node_compile_binary_function (en, pn, var_list)
  case ("eval_fun")
     call eval_node_compile_eval_function (en, pn, var_list)
  case ("count_fun")
     call eval_node_compile_int_function (en, pn, var_list)
  case default
     call parse_node_mismatch &
          ("integer|real|complex|constant|variable|" // &
           "expr|block_expr|conditional_expr|" // &
           "unary_function|binary_function|numeric_pexpr", pn)
  end select
  if (debug) then
     call eval_node_write (en)
     print *, "done value"
  end if
end subroutine eval_node_compile_value
```

Real, complex and integer values are numeric literals with an optional unit attached. In case of an integer, the unit actually makes it a real value in disguise. The signed version of real values is not possible in generic expressions; it is a special case for numeric constants in model files (see below). We do not introduce signed versions of complex values.

⟨*Expressions: procedures*⟩+≡

```
subroutine eval_node_compile_numeric_value (en, pn)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in), target :: pn
```

```
      type(parse_node_t), pointer :: pn_val, pn_unit
      allocate (en)
      pn_val => parse_node_get_sub_ptr (pn)
      pn_unit => parse_node_get_next_ptr (pn_val)
      select case (char (parse_node_get_rule_key (pn)))
      case ("integer_value")
         if (associated (pn_unit)) then
            call eval_node_init_real (en, &
                 parse_node_get_integer (pn_val) * parse_node_get_unit (pn_unit))
         else
            call eval_node_init_int (en, parse_node_get_integer (pn_val))
         end if
      case ("real_value")
         if (associated (pn_unit)) then
            call eval_node_init_real (en, &
                 parse_node_get_real (pn_val) * parse_node_get_unit (pn_unit))
         else
            call eval_node_init_real (en, parse_node_get_real (pn_val))
         end if
      case ("complex_value")
         if (associated (pn_unit)) then
            call eval_node_init_cmplx (en, &
                 parse_node_get_cmplx (pn_val) * parse_node_get_unit (pn_unit))
         else
            call eval_node_init_cmplx (en, parse_node_get_cmplx (pn_val))
         end if
      case ("neg_real_value")
         pn_val => parse_node_get_sub_ptr (parse_node_get_sub_ptr (pn, 2))
         pn_unit => parse_node_get_next_ptr (pn_val)
         if (associated (pn_unit)) then
            call eval_node_init_real (en, &
                 - parse_node_get_real (pn_val) * parse_node_get_unit (pn_unit))
         else
            call eval_node_init_real (en, - parse_node_get_real (pn_val))
         end if
      case ("pos_real_value")
         pn_val => parse_node_get_sub_ptr (parse_node_get_sub_ptr (pn, 2))
         pn_unit => parse_node_get_next_ptr (pn_val)
         if (associated (pn_unit)) then
            call eval_node_init_real (en, &
                 parse_node_get_real (pn_val) * parse_node_get_unit (pn_unit))
         else
            call eval_node_init_real (en, parse_node_get_real (pn_val))
         end if
      case default
         call parse_node_mismatch &
         ("integer_value|real_value|complex_value|neg_real_value|pos_real_value", pn)
      end select
    end subroutine eval_node_compile_numeric_value
```

These are the units, predefined and hardcoded. The default energy unit is
GeV, the default angular unit is radians. We include units for observables of
dimension energy squared. Luminosities are normalized in inverse femtobarns.

382

⟨*Expressions: procedures*⟩+≡

```
function parse_node_get_unit (pn) result (factor)
  real(default) :: factor
  real(default) :: unit
  type(parse_node_t), intent(in) :: pn
  type(parse_node_t), pointer :: pn_unit, pn_unit_power
  type(parse_node_t), pointer :: pn_frac, pn_num, pn_int, pn_div, pn_den
  integer :: num, den
  pn_unit => parse_node_get_sub_ptr (pn)
  select case (char (parse_node_get_key (pn_unit)))
  case ("TeV");   unit = 1.e3_default
  case ("GeV");   unit = 1
  case ("MeV");   unit = 1.e-3_default
  case ("keV");   unit = 1.e-6_default
  case ("eV");    unit = 1.e-9_default
  case ("meV");    unit = 1.e-12_default
  case ("nbarn");  unit = 1.e6_default
  case ("pbarn");  unit = 1.e3_default
  case ("fbarn");  unit = 1
  case ("abarn");  unit = 1.e-3_default
  case ("rad");     unit = 1
  case ("mrad");    unit = 1.e-3_default
  case ("degree");  unit = degree
  case ("%");  unit = 1.e-2_default
  case default
     call msg_bug (" Unit '" // &
          char (parse_node_get_key (pn)) // "' is undefined.")
  end select
  pn_unit_power => parse_node_get_next_ptr (pn_unit)
  if (associated (pn_unit_power)) then
     pn_frac => parse_node_get_sub_ptr (pn_unit_power, 2)
     pn_num => parse_node_get_sub_ptr (pn_frac)
     select case (char (parse_node_get_rule_key (pn_num)))
     case ("neg_int")
        pn_int => parse_node_get_sub_ptr (pn_num, 2)
        num = - parse_node_get_integer (pn_int)
     case ("pos_int")
        pn_int => parse_node_get_sub_ptr (pn_num, 2)
        num = parse_node_get_integer (pn_int)
     case ("integer_literal")
        num = parse_node_get_integer (pn_num)
     case default
        call parse_node_mismatch ("neg_int|pos_int|integer_literal", pn_num)
     end select
     pn_div => parse_node_get_next_ptr (pn_num)
     if (associated (pn_div)) then
        pn_den => parse_node_get_sub_ptr (pn_div, 2)
        den = parse_node_get_integer (pn_den)
     else
        den = 1
     end if
  else
     num = 1
     den = 1
```

```
         end if
         factor = unit ** (real (num, default) / den)
      end function parse_node_get_unit
```

There are only two predefined constants, but more can be added easily.

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_node_compile_constant (en, pn)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      if (debug) then
         print *, "read constant";  call parse_node_write (pn)
      end if
      allocate (en)
      select case (char (parse_node_get_key (pn)))
      case ("pi");     call eval_node_init_real (en, pi)
      case ("I");      call eval_node_init_cmplx (en, imago)
      case default
         call parse_node_mismatch ("pi or I", pn)
      end select
      if (debug) then
         call eval_node_write (en)
         print *, "done constant"
      end if
    end subroutine eval_node_compile_constant
```

Compile a variable, with or without a specified type. Take the list of variables, look for the name and make a node with a pointer to the value. If no type is provided, the variable is numeric, and the stored value determines whether it is real or integer.

Variables may actually be anonymous, these are expressions in disguise. In that case, the expression replaces the variable name in the parse tree, and we allocate an ordinary expression node in the eval tree.

Variables of type `V_PDG` (pdg-code array) are not treated here. They are handled by `eval_node_compile_cvariable`.

⟨*Expressions: procedures*⟩+≡
```
    recursive subroutine eval_node_compile_variable (en, pn, var_list, var_type)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in), target :: pn
      type(var_list_t), intent(in), target :: var_list
      integer, intent(in), optional :: var_type
      type(parse_node_t), pointer :: pn_name
      type(string_t) :: var_name
      type(var_entry_t), pointer :: var
      logical, target, save :: no_lval
      real(default), target, save :: no_rval
      type(prt_list_t), target, save :: no_pval
      type(string_t), target, save :: no_sval
      logical, target, save :: unknown = .false.
      if (debug) then
         print *, "read variable";  call parse_node_write (pn)
      end if
      if (present (var_type)) then
```

```
          select case (var_type)
          case (V_REAL, V_OBS1_REAL, V_OBS2_REAL, V_INT, V_OBS1_INT, &
                  V_OBS2_INT, V_CMPLX)
             pn_name => pn
          case default
             pn_name => parse_node_get_sub_ptr (pn, 2)
          end select
       else
          pn_name => pn
       end if
       select case (char (parse_node_get_rule_key (pn_name)))
       case ("expr")
          call eval_node_compile_expr (en, pn_name, var_list)
       case ("lexpr")
          call eval_node_compile_lexpr (en, pn_name, var_list)
       case ("sexpr")
          call eval_node_compile_sexpr (en, pn_name, var_list)
       case ("pexpr")
          call eval_node_compile_pexpr (en, pn_name, var_list)
       case ("variable")
          var_name = parse_node_get_string (pn_name)
          if (present (var_type)) then
             select case (var_type)
             case (V_LOG);  var_name = "?" // var_name
             case (V_PTL);  var_name = "@" // var_name
             case (V_STR);  var_name = "$" // var_name   ! $ sign
             end select
          end if
          var => var_list_get_var_ptr (var_list, var_name, var_type)
          allocate (en)
          if (associated (var)) then
             select case (var_entry_get_type (var))
             case (V_LOG)
                call eval_node_init_log_ptr &
                     (en, var_entry_get_name (var), var_entry_get_lval_ptr (var), &
                      var_entry_get_known_ptr (var))
             case (V_INT)
                call eval_node_init_int_ptr &
                     (en, var_entry_get_name (var), var_entry_get_ival_ptr (var), &
                      var_entry_get_known_ptr (var))
             case (V_REAL)
                call eval_node_init_real_ptr &
                     (en, var_entry_get_name (var), var_entry_get_rval_ptr (var), &
                      var_entry_get_known_ptr (var))
             case (V_CMPLX)
                call eval_node_init_cmplx_ptr &
                     (en, var_entry_get_name (var), var_entry_get_cval_ptr (var), &
                      var_entry_get_known_ptr (var))
             case (V_PTL)
                call eval_node_init_prt_list_ptr &
                     (en, var_entry_get_name (var), var_entry_get_pval_ptr (var), &
                      var_entry_get_known_ptr (var))
             case (V_STR)
                call eval_node_init_string_ptr &
```

```
                    (en, var_entry_get_name (var), var_entry_get_sval_ptr (var), &
                     var_entry_get_known_ptr (var))
            case (V_OBS1_INT, V_OBS2_INT, V_OBS1_REAL, V_OBS2_REAL)
               call eval_node_init_obs  (en, var)
            case default
               call parse_node_write (pn)
               call msg_fatal ("Variable of this type " // &
                    "is not allowed in the present context")
               if (present (var_type)) then
                  select case (var_type)
                  case (V_LOG)
                     call eval_node_init_log_ptr (en, var_name, no_lval, unknown)
                  case (V_PTL)
                     call eval_node_init_prt_list_ptr &
                          (en, var_name, no_pval, unknown)
                  case (V_STR)
                     call eval_node_init_string_ptr &
                          (en, var_name, no_sval, unknown)
                  end select
               else
                  call eval_node_init_real_ptr (en, var_name, no_rval, unknown)
               end if
            end select
         else
            call parse_node_write (pn)
            call msg_error ("This variable is undefined at this point")
            if (present (var_type)) then
               select case (var_type)
               case (V_LOG)
                  call eval_node_init_log_ptr (en, var_name, no_lval, unknown)
               case (V_PTL)
                  call eval_node_init_prt_list_ptr &
                       (en, var_name, no_pval, unknown)
               case (V_STR)
                  call eval_node_init_string_ptr (en, var_name, no_sval, unknown)
               end select
            else
               call eval_node_init_real_ptr (en, var_name, no_rval, unknown)
            end if
         end if
      end select
      if (debug) then
         call eval_node_write (en)
         print *, "done variable"
      end if
    end subroutine eval_node_compile_variable
```

In a given context, a variable has to have a certain type.

⟨*Expressions: procedures*⟩+≡
```
    subroutine check_var_type (pn, ok, type_actual, type_requested)
      type(parse_node_t), intent(in) :: pn
      logical, intent(out) :: ok
      integer, intent(in) :: type_actual
      integer, intent(in), optional :: type_requested
```

```
      if (present (type_requested)) then
         select case (type_requested)
         case (V_LOG)
            select case (type_actual)
            case (V_LOG)
            case default
               call parse_node_write (pn)
               call msg_fatal ("Variable type is invalid (should be logical)")
               ok = .false.
            end select
         case (V_PTL)
            select case (type_actual)
            case (V_PTL)
            case default
               call parse_node_write (pn)
               call msg_fatal &
                    ("Variable type is invalid (should be particle set)")
               ok = .false.
            end select
         case (V_PDG)
            select case (type_actual)
            case (V_PDG)
            case default
               call parse_node_write (pn)
               call msg_fatal &
                    ("Variable type is invalid (should be PDG array)")
               ok = .false.
            end select
         case (V_STR)
            select case (type_actual)
            case (V_STR)
            case default
               call parse_node_write (pn)
               call msg_fatal &
                    ("Variable type is invalid (should be string)")
               ok = .false.
            end select
         case default
            call parse_node_write (pn)
            call msg_bug ("Variable type is unknown")
         end select
      else
         select case (type_actual)
         case (V_REAL, V_OBS1_REAL, V_OBS2_REAL, V_INT, V_OBS1_INT, &
                 V_OBS2_INT, V_CMPLX)
         case default
            call parse_node_write (pn)
            call msg_fatal ("Variable type is invalid (should be numeric)")
            ok = .false.
         end select
      end if
      ok = .true.
    end subroutine check_var_type
```

Retrieve the result of an integration. If the requested process has been integrated, the results are available as special variables. (The variables cannot be accessed in the usual way since they contain brackets in their names.)

Since this compilation step may occur before the processes have been loaded, we have to initialize the required variables before they are used.

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_node_compile_result (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in), target :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_key, pn_prc_id
      type(string_t) :: key, prc_id, var_name
      type(var_entry_t), pointer :: var
      if (debug) then
         print *, "read result";  call parse_node_write (pn)
      end if
      pn_key => parse_node_get_sub_ptr (pn)
      pn_prc_id => parse_node_get_next_ptr (pn_key)
      key = parse_node_get_key (pn_key)
      prc_id = parse_node_get_string (pn_prc_id)
      var_name = key // "(" // prc_id // ")"
      var => var_list_get_var_ptr (var_list, var_name)
      if (associated (var)) then
         allocate (en)
         select case (char(key))
         case ("n_calls")
            call eval_node_init_int_ptr &
                 (en, var_name, var_entry_get_ival_ptr (var), &
                 var_entry_get_known_ptr (var))
         case ("integral", "error", "accuracy", "chi2", "efficiency")
            call eval_node_init_real_ptr &
                 (en, var_name, var_entry_get_rval_ptr (var), &
               var_entry_get_known_ptr (var))
         end select
      else
         call msg_error ("Result variable '" // char (var_name) &
              // "' is undefined (call 'integrate' before use)")
      end if
      if (debug) then
         call eval_node_write (en)
         print *, "done result"
      end if
    end subroutine eval_node_compile_result
```

Functions with a single argument. For non-constant arguments, watch for functions which convert their argument to a different type.

⟨*Expressions: procedures*⟩+≡
```
    recursive subroutine eval_node_compile_unary_function (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_fname, pn_arg
      type(eval_node_t), pointer :: en1
```

```
type(string_t) :: key
integer :: t
if (debug) then
   print *, "read unary function";  call parse_node_write (pn)
end if
pn_fname => parse_node_get_sub_ptr (pn)
pn_arg => parse_node_get_next_ptr (pn_fname, tag="function_arg1")
call eval_node_compile_expr &
     (en1, parse_node_get_sub_ptr (pn_arg, tag="expr"), var_list)
t = en1%result_type
allocate (en)
key = parse_node_get_key (pn_fname)
if (en1%type == EN_CONSTANT) then
   select case (char (key))
   case ("real")
      select case (t)
      case (V_INT);  call eval_node_init_real (en, real_i (en1))
      case (V_REAL); deallocate (en); en => en1
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("int")
      select case (t)
      case (V_INT);  deallocate (en);  en => en1
      case (V_REAL); call eval_node_init_int  (en, int_r (en1))
      case (V_CMPLX); call eval_node_init_int  (en, int_c (en1))
      end select
   case ("nint")
      select case (t)
      case (V_INT);  deallocate (en);  en => en1
      case (V_REAL); call eval_node_init_int  (en, nint_r (en1))
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("floor")
      select case (t)
      case (V_INT);  deallocate (en);  en => en1
      case (V_REAL); call eval_node_init_int  (en, floor_r (en1))
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("ceiling")
      select case (t)
      case (V_INT);  deallocate (en);  en => en1
      case (V_REAL); call eval_node_init_int  (en, ceiling_r (en1))
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("abs")
      select case (t)
      case (V_INT);  call eval_node_init_int  (en, abs_i (en1))
      case (V_REAL); call eval_node_init_real (en, abs_r (en1))
      case (V_CMPLX); call eval_node_init_real (en, abs_c (en1))
      end select
   case ("sgn")
      select case (t)
      case (V_INT);  call eval_node_init_int  (en, sgn_i (en1))
      case (V_REAL); call eval_node_init_real (en, sgn_r (en1))
```

```
      case default;  call eval_type_error (pn, char (key), t)
      end select
case ("sqrt")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, sqrt_r (en1))
   case (V_CMPLX); call eval_node_init_cmplx (en, sqrt_c (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("exp")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, exp_r (en1))
   case (V_CMPLX); call eval_node_init_cmplx (en, exp_c (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("log")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, log_r (en1))
   case (V_CMPLX); call eval_node_init_cmplx (en, log_c (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("log10")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, log10_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("sin")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, sin_r (en1))
   case (V_CMPLX); call eval_node_init_cmplx (en, sin_c (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("cos")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, cos_r (en1))
   case (V_CMPLX); call eval_node_init_cmplx (en, cos_c (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("tan")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, tan_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("asin")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, asin_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("acos")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, acos_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("atan")
   select case (t)
```

```
              case (V_REAL); call eval_node_init_real (en, atan_r (en1))
              case default;  call eval_type_error (pn, char (key), t)
              end select
           case ("sinh")
              select case (t)
              case (V_REAL); call eval_node_init_real (en, sinh_r (en1))
              case default;  call eval_type_error (pn, char (key), t)
              end select
           case ("cosh")
              select case (t)
              case (V_REAL); call eval_node_init_real (en, cosh_r (en1))
              case default;  call eval_type_error (pn, char (key), t)
              end select
           case ("tanh")
              select case (t)
              case (V_REAL); call eval_node_init_real (en, tanh_r (en1))
              case default;  call eval_type_error (pn, char (key), t)
              end select
           case default
              call parse_node_mismatch ("function name", pn_fname)
           end select
           call eval_node_final_rec (en1)
           deallocate (en1)
        else
           select case (char (key))
           case ("real")
              call eval_node_init_branch (en, key, V_REAL, en1)
           case ("int", "nint", "floor", "ceiling")
              call eval_node_init_branch (en, key, V_INT, en1)
           case default
              call eval_node_init_branch (en, key, t, en1)
           end select
           select case (char (key))
           case ("real")
              select case (t)
              case (V_INT);  call eval_node_set_op1_real (en, real_i)
              case (V_REAL); deallocate (en); en => en1
              case default;  call eval_type_error (pn, char (key), t)
              end select
           case ("int")
              select case (t)
              case (V_INT);  deallocate (en);  en => en1
              case (V_REAL); call eval_node_set_op1_int (en, int_r)
              case (V_CMPLX); call eval_node_set_op1_int (en, int_c)
              end select
           case ("nint")
              select case (t)
              case (V_INT);  deallocate (en);  en => en1
              case (V_REAL); call eval_node_set_op1_int (en, nint_r)
              case default;  call eval_type_error (pn, char (key), t)
              end select
           case ("floor")
              select case (t)
              case (V_INT);  deallocate (en);  en => en1
```

```
      case (V_REAL); call eval_node_set_op1_int (en, floor_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("ceiling")
      select case (t)
      case (V_INT);  deallocate (en);  en => en1
      case (V_REAL); call eval_node_set_op1_int (en, ceiling_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("abs")
      select case (t)
      case (V_INT);  call eval_node_set_op1_int  (en, abs_i)
      case (V_REAL); call eval_node_set_op1_real (en, abs_r)
      case (V_CMPLX); call eval_node_set_op1_real (en, abs_c)
      end select
   case ("sgn")
      select case (t)
      case (V_INT);  call eval_node_set_op1_int  (en, sgn_i)
      case (V_REAL); call eval_node_set_op1_real (en, sgn_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("sqrt")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, sqrt_r)
      case (V_CMPLX); call eval_node_set_op1_cmplx (en, sqrt_c)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("exp")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, exp_r)
      case (V_CMPLX); call eval_node_set_op1_cmplx (en, exp_c)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("log")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, log_r)
      case (V_CMPLX); call eval_node_set_op1_cmplx (en, log_c)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("log10")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, log10_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("sin")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, sin_r)
      case (V_CMPLX); call eval_node_set_op1_cmplx (en, sin_c)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("cos")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, cos_r)
      case (V_CMPLX); call eval_node_set_op1_cmplx (en, cos_c)
```

```
              case default;  call eval_type_error (pn, char (key), t)
              end select
         case ("tan")
            select case (t)
            case (V_REAL); call eval_node_set_op1_real (en, tan_r)
            case default;  call eval_type_error (pn, char (key), t)
            end select
         case ("asin")
            select case (t)
            case (V_REAL); call eval_node_set_op1_real (en, asin_r)
            case default;  call eval_type_error (pn, char (key), t)
            end select
         case ("acos")
            select case (t)
            case (V_REAL); call eval_node_set_op1_real (en, acos_r)
            case default;  call eval_type_error (pn, char (key), t)
            end select
         case ("atan")
            select case (t)
            case (V_REAL); call eval_node_set_op1_real (en, atan_r)
            case default;  call eval_type_error (pn, char (key), t)
            end select
         case ("sinh")
            select case (t)
            case (V_REAL); call eval_node_set_op1_real (en, sinh_r)
            case default;  call eval_type_error (pn, char (key), t)
            end select
         case ("cosh")
            select case (t)
            case (V_REAL); call eval_node_set_op1_real (en, cosh_r)
            case default;  call eval_type_error (pn, char (key), t)
            end select
         case ("tanh")
            select case (t)
            case (V_REAL); call eval_node_set_op1_real (en, tanh_r)
            case default;  call eval_type_error (pn, char (key), t)
            end select
         case default
            call parse_node_mismatch ("function name", pn_fname)
         end select
      end if
      if (debug) then
         call eval_node_write (en)
         print *, "done function"
      end if
    end subroutine eval_node_compile_unary_function
```

Functions with two arguments.

⟨*Expressions: procedures*⟩+≡

```
    recursive subroutine eval_node_compile_binary_function (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_fname, pn_arg, pn_arg1, pn_arg2
```

```
type(eval_node_t), pointer :: en1, en2
type(string_t) :: key
integer :: t1, t2
if (debug) then
   print *, "read binary function";  call parse_node_write (pn)
end if
pn_fname => parse_node_get_sub_ptr (pn)
pn_arg => parse_node_get_next_ptr (pn_fname, tag="function_arg2")
pn_arg1 => parse_node_get_sub_ptr (pn_arg, tag="expr")
pn_arg2 => parse_node_get_next_ptr (pn_arg1, tag="expr")
call eval_node_compile_expr (en1, pn_arg1, var_list)
call eval_node_compile_expr (en2, pn_arg2, var_list)
t1 = en1%result_type
t2 = en2%result_type
allocate (en)
key = parse_node_get_key (pn_fname)
if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
   select case (char (key))
   case ("max")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_int  (en, max_ii (en1, en2))
         case (V_REAL); call eval_node_init_real (en, max_ir (en1, en2))
         case default;  call eval_type_error (pn, char (key), t2)
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_real (en, max_ri (en1, en2))
         case (V_REAL); call eval_node_init_real (en, max_rr (en1, en2))
         case default;  call eval_type_error (pn, char (key), t2)
         end select
       case default;  call eval_type_error (pn, char (key), t1)
     end select
   case ("min")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_int  (en, min_ii (en1, en2))
         case (V_REAL); call eval_node_init_real (en, min_ir (en1, en2))
         case default;  call eval_type_error (pn, char (key), t2)
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_real (en, min_ri (en1, en2))
         case (V_REAL); call eval_node_init_real (en, min_rr (en1, en2))
         case default;  call eval_type_error (pn, char (key), t2)
         end select
       case default;  call eval_type_error (pn, char (key), t1)
     end select
   case ("mod")
      select case (t1)
      case (V_INT)
         select case (t2)
```

```
              case (V_INT);  call eval_node_init_int  (en, mod_ii (en1, en2))
              case (V_REAL); call eval_node_init_real (en, mod_ir (en1, en2))
              case default;  call eval_type_error (pn, char (key), t2)
              end select
         case (V_REAL)
            select case (t2)
            case (V_INT);  call eval_node_init_real (en, mod_ri (en1, en2))
            case (V_REAL); call eval_node_init_real (en, mod_rr (en1, en2))
            case default;  call eval_type_error (pn, char (key), t2)
            end select
          case default;  call eval_type_error (pn, char (key), t1)
        end select
      case ("modulo")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);  call eval_node_init_int  (en, modulo_ii (en1, en2))
            case (V_REAL); call eval_node_init_real (en, modulo_ir (en1, en2))
            case default;  call eval_type_error (pn, char (key), t2)
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);  call eval_node_init_real (en, modulo_ri (en1, en2))
            case (V_REAL); call eval_node_init_real (en, modulo_rr (en1, en2))
            case default;  call eval_type_error (pn, char (key), t2)
            end select
          case default;  call eval_type_error (pn, char (key), t2)
        end select
      case default
         call parse_node_mismatch ("function name", pn_fname)
      end select
      call eval_node_final_rec (en1)
      deallocate (en1)
   else
      call eval_node_init_branch (en, key, t1, en1, en2)
      select case (char (key))
      case ("max")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_int  (en, max_ii)
            case (V_REAL); call eval_node_set_op2_real (en, max_ir)
            case default;  call eval_type_error (pn, char (key), t2)
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_real (en, max_ri)
            case (V_REAL); call eval_node_set_op2_real (en, max_rr)
            case default;  call eval_type_error (pn, char (key), t2)
            end select
          case default;  call eval_type_error (pn, char (key), t2)
        end select
      case ("min")
         select case (t1)
```

```
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_set_op2_int  (en, min_ii)
         case (V_REAL); call eval_node_set_op2_real (en, min_ir)
         case default;  call eval_type_error (pn, char (key), t2)
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_set_op2_real (en, min_ri)
         case (V_REAL); call eval_node_set_op2_real (en, min_rr)
         case default;  call eval_type_error (pn, char (key), t2)
         end select
       case default;  call eval_type_error (pn, char (key), t2)
     end select
   case ("mod")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_set_op2_int  (en, mod_ii)
         case (V_REAL); call eval_node_set_op2_real (en, mod_ir)
         case default;  call eval_type_error (pn, char (key), t2)
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_set_op2_real (en, mod_ri)
         case (V_REAL); call eval_node_set_op2_real (en, mod_rr)
         case default;  call eval_type_error (pn, char (key), t2)
         end select
       case default;  call eval_type_error (pn, char (key), t2)
     end select
   case ("modulo")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_set_op2_int  (en, modulo_ii)
         case (V_REAL); call eval_node_set_op2_real (en, modulo_ir)
         case default;  call eval_type_error (pn, char (key), t2)
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_set_op2_real (en, modulo_ri)
         case (V_REAL); call eval_node_set_op2_real (en, modulo_rr)
         case default;  call eval_type_error (pn, char (key), t2)
         end select
       case default;  call eval_type_error (pn, char (key), t2)
     end select
   case default
      call parse_node_mismatch ("function name", pn_fname)
   end select
end if
if (debug) then
   call eval_node_write (en)
   print *, "done function"
end if
```

```
    end subroutine eval_node_compile_binary_function
```

Variable definition

A block expression contains a variable definition (first argument) and an expression where the definition can be used (second argument). The `result_type` decides which type of expression is expected for the second argument. For numeric variables, if there is a mismatch between real and integer type, insert an extra node for type conversion.

⟨*Expressions: procedures*⟩+≡
```
    recursive subroutine eval_node_compile_block_expr &
        (en, pn, var_list, result_type)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      integer, intent(in), optional :: result_type
      type(parse_node_t), pointer :: pn_var_spec, pn_var_subspec
      type(parse_node_t), pointer :: pn_var_type, pn_var_name, pn_var_expr
      type(parse_node_t), pointer :: pn_expr
      type(string_t) :: var_name
      type(eval_node_t), pointer :: en1, en2
      integer :: var_type
      logical :: new
      if (debug) then
         print *, "read block expr";  call parse_node_write (pn)
      end if
      new = .false.
      pn_var_spec => parse_node_get_sub_ptr (pn, 2)
      select case (char (parse_node_get_rule_key (pn_var_spec)))
      case ("var_num");       var_type = V_NONE
         pn_var_name => parse_node_get_sub_ptr (pn_var_spec)
      case ("var_int");       var_type = V_INT
         new = .true.
         pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
      case ("var_real");      var_type = V_REAL
         new = .true.
         pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
      case ("var_cmplx");     var_type = V_CMPLX
         new = .true.
         pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
      case ("var_logical_new");  var_type = V_LOG
         new = .true.
         pn_var_subspec => parse_node_get_sub_ptr (pn_var_spec, 2)
         pn_var_name => parse_node_get_sub_ptr (pn_var_subspec, 2)
      case ("var_logical_spec");  var_type = V_LOG
         pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
      case ("var_plist_new");     var_type = V_PTL
         new = .true.
         pn_var_subspec => parse_node_get_sub_ptr (pn_var_spec, 2)
         pn_var_name => parse_node_get_sub_ptr (pn_var_subspec, 2)
      case ("var_plist_spec");    var_type = V_PTL
         new = .true.
```

397

```
      pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
   case ("var_alias");    var_type = V_PDG
      new = .true.
      pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
   case ("var_string_new");   var_type = V_STR
      new = .true.
      pn_var_subspec => parse_node_get_sub_ptr (pn_var_spec, 2)
      pn_var_name => parse_node_get_sub_ptr (pn_var_subspec, 2)
   case ("var_string_spec");   var_type = V_STR
      pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
   case default
      call parse_node_mismatch &
           ("logical|int|real|plist|alias", pn_var_type)
   end select
   pn_var_expr => parse_node_get_next_ptr (pn_var_name, 2)
   pn_expr => parse_node_get_next_ptr (pn_var_spec, 2)
   var_name = parse_node_get_string (pn_var_name)
   select case (var_type)
   case (V_LOG);  var_name = "?" // var_name
   case (V_PTL);  var_name = "@" // var_name
   case (V_STR);  var_name = "$" // var_name    ! $ sign
   end select
   call var_list_check_user_var (var_list, var_name, var_type, new)
   call eval_node_compile_genexpr (en1, pn_var_expr, var_list, var_type)
   call insert_conversion_node (en1, var_type)
   allocate (en)
   call eval_node_init_block (en, var_name, var_type, en1, var_list)
   call eval_node_compile_genexpr (en2, pn_expr, en%var_list, result_type)
   call eval_node_set_expr (en, en2)
   if (debug) then
      call eval_node_write (en)
      print *, "done block expr"
   end if
end subroutine eval_node_compile_block_expr
```

Insert a conversion node for integer/real/complex transformation if necessary.
What shall we do for the complex to integer/real conversion?

⟨*Expressions: procedures*⟩+≡
```
subroutine insert_conversion_node (en, result_type)
  type(eval_node_t), pointer :: en
  integer, intent(in) :: result_type
  type(eval_node_t), pointer :: en_conv
  select case (en%result_type)
  case (V_INT)
     select case (result_type)
     case (V_REAL)
        allocate (en_conv)
        call eval_node_init_branch (en_conv, var_str ("real"), V_REAL, en)
        call eval_node_set_op1_real (en_conv, real_i)
        en => en_conv
     case (V_CMPLX)
        allocate (en_conv)
        call eval_node_init_branch (en_conv, var_str ("complex"), V_CMPLX, en)
```

```
               call eval_node_set_op1_cmplx (en_conv, cmplx_i)
               en => en_conv
            end select
         case (V_REAL)
            select case (result_type)
            case (V_INT)
               allocate (en_conv)
               call eval_node_init_branch (en_conv, var_str ("int"), V_INT, en)
               call eval_node_set_op1_int (en_conv, int_r)
               en => en_conv
            case (V_CMPLX)
               allocate (en_conv)
               call eval_node_init_branch (en_conv, var_str ("complex"), V_CMPLX, en)
               call eval_node_set_op1_cmplx (en_conv, cmplx_r)
               en => en_conv
            end select
         case (V_CMPLX)
            select case (result_type)
            case (V_INT)
               allocate (en_conv)
               call eval_node_init_branch (en_conv, var_str ("int"), V_INT, en)
               call eval_node_set_op1_int (en_conv, int_c)
               en => en_conv
            case (V_REAL)
               allocate (en_conv)
               call eval_node_init_branch (en_conv, var_str ("real"), V_REAL, en)
               call eval_node_set_op1_real (en_conv, real_c)
               en => en_conv
            end select
         case default
         end select
      end subroutine insert_conversion_node
```

Conditionals

A conditional has the structure if lexpr then expr else expr. So we first eval-
uate the logical expression, then depending on the result the first or second
expression. Note that the second expression is mandatory.

The `result_type`, if present, defines the requested type of the **then** and
else clauses. Default is numeric (int/real). If there is a mismatch between real
and integer result types, insert conversion nodes.

⟨*Expressions: procedures*⟩+≡
```
   recursive subroutine eval_node_compile_conditional &
       (en, pn, var_list, result_type)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     integer, intent(in), optional :: result_type
     type(parse_node_t), pointer :: pn_condition, pn_expr1, pn_expr2
     type(eval_node_t), pointer :: en0, en1, en2
     integer :: t1, t2
     if (debug) then
```

```
      print *, "read conditional";  call parse_node_write (pn)
   end if
   pn_condition => parse_node_get_sub_ptr (pn, 2, tag="lexpr")
   pn_expr1 => parse_node_get_next_ptr (pn_condition, 2)
   pn_expr2 => parse_node_get_next_ptr (pn_expr1, 2)
   call eval_node_compile_lexpr (en0, pn_condition, var_list)
   call eval_node_compile_genexpr (en1, pn_expr1, var_list, result_type)
   call eval_node_compile_genexpr (en2, pn_expr2, var_list, result_type)
   t1 = en1%result_type
   t2 = en2%result_type
   if (t1 == t2) then
      if (en0%type == EN_CONSTANT) then
         if (en0%lval) then
            en => en1
            call eval_node_final_rec (en2)
            deallocate (en2)
         else
            en => en2
            call eval_node_final_rec (en1)
            deallocate (en1)
         end if
         call eval_node_final_rec (en0)
         deallocate (en0)
      else
         allocate (en)
         call eval_node_init_conditional (en, t1, en0, en1, en2)
      end if
   else if (t1 == V_INT) then
      call insert_conversion_node (en1, V_REAL)
   else if (t2 == V_INT) then
      call insert_conversion_node (en2, V_REAL)
   else
      call parse_node_write (pn)
      call msg_bug &
           (" The 'then' and 'else' expressions have incompatible types")
   end if
   if (debug) then
      call eval_node_write (en)
      print *, "done conditional"
   end if
end subroutine eval_node_compile_conditional
```

Logical expressions

A logical expression consists of one or more logical terms concatenated by **or**.

⟨*Expressions: procedures*⟩+≡

```
recursive subroutine eval_node_compile_lexpr (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_term, pn_alternative, pn_arg
  type(eval_node_t), pointer :: en1, en2
  if (debug) then
```

```
         print *, "read lexpr";  call parse_node_write (pn)
      end if
      pn_term => parse_node_get_sub_ptr (pn, tag="lterm")
      call eval_node_compile_lterm (en, pn_term, var_list)
      pn_alternative => parse_node_get_next_ptr (pn_term, tag="alternative")
      do while (associated (pn_alternative))
         pn_arg => parse_node_get_sub_ptr (pn_alternative, 2, tag="lterm")
         en1 => en
         call eval_node_compile_lterm (en2, pn_arg, var_list)
         allocate (en)
         if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
            call eval_node_init_log (en, or_ll (en1, en2))
            call eval_node_final_rec (en1)
            call eval_node_final_rec (en2)
            deallocate (en1, en2)
         else
            call eval_node_init_branch &
                 (en, var_str ("alternative"), V_LOG, en1, en2)
            call eval_node_set_op2_log (en, or_ll)
         end if
         pn_alternative => parse_node_get_next_ptr (pn_alternative)
      end do
      if (debug) then
         call eval_node_write (en)
         print *, "done lexpr"
      end if
   end subroutine eval_node_compile_lexpr
```

A logical term consists of one or more logical values concatenated by and.
⟨*Expressions: procedures*⟩+≡

```
   recursive subroutine eval_node_compile_lterm (en, pn, var_list)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     type(parse_node_t), pointer :: pn_term, pn_coincidence, pn_arg
     type(eval_node_t), pointer :: en1, en2
     if (debug) then
        print *, "read lterm";  call parse_node_write (pn)
     end if
     pn_term => parse_node_get_sub_ptr (pn)
     call eval_node_compile_lvalue (en, pn_term, var_list)
     pn_coincidence => parse_node_get_next_ptr (pn_term, tag="coincidence")
     do while (associated (pn_coincidence))
        pn_arg => parse_node_get_sub_ptr (pn_coincidence, 2)
        en1 => en
        call eval_node_compile_lvalue (en2, pn_arg, var_list)
        allocate (en)
        if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
           call eval_node_init_log (en, and_ll (en1, en2))
           call eval_node_final_rec (en1)
           call eval_node_final_rec (en2)
           deallocate (en1, en2)
        else
           call eval_node_init_branch &
```

```
                (en, var_str ("coincidence"), V_LOG, en1, en2)
            call eval_node_set_op2_log (en, and_ll)
         end if
         pn_coincidence => parse_node_get_next_ptr (pn_coincidence)
      end do
      if (debug) then
         call eval_node_write (en)
         print *, "done lterm"
      end if
   end subroutine eval_node_compile_lterm
```

Logical variables are disabled, because they are confused with the l.h.s. of compared expressions.

⟨*Expressions: procedures*⟩+≡

```
   recursive subroutine eval_node_compile_lvalue (en, pn, var_list)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     if (debug) then
        print *, "read lvalue";  call parse_node_write (pn)
     end if
     select case (char (parse_node_get_rule_key (pn)))
     case ("true")
        allocate (en)
        call eval_node_init_log (en, .true.)
     case ("false")
        allocate (en)
        call eval_node_init_log (en, .false.)
     case ("negation")
        call eval_node_compile_negation (en, pn, var_list)
     case ("lvariable")
        call eval_node_compile_variable (en, pn, var_list, V_LOG)
     case ("lexpr")
        call eval_node_compile_lexpr (en, pn, var_list)
     case ("block_lexpr")
        call eval_node_compile_block_expr (en, pn, var_list, V_LOG)
     case ("conditional_lexpr")
        call eval_node_compile_conditional (en, pn, var_list, V_LOG)
     case ("compared_expr")
        call eval_node_compile_compared_expr (en, pn, var_list, V_REAL)
     case ("compared_sexpr")
        call eval_node_compile_compared_expr (en, pn, var_list, V_STR)
     case ("all_fun", "any_fun", "no_fun")
        call eval_node_compile_log_function (en, pn, var_list)
     case ("record_cmd")
        call eval_node_compile_record_cmd (en, pn, var_list)
     case default
        call parse_node_mismatch &
             ("true|false|negation|lvariable|" // &
              "lexpr|block_lexpr|conditional_lexpr|" // &
              "compared_expr|compared_sexpr|logical_pexpr", pn)
     end select
     if (debug) then
```

```
          call eval_node_write (en)
          print *, "done lvalue"
        end if
    end subroutine eval_node_compile_lvalue
```

A negation consists of the keyword **not** and a logical value.

⟨*Expressions: procedures*⟩+≡
```
    recursive subroutine eval_node_compile_negation (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_arg
      type(eval_node_t), pointer :: en1
      if (debug) then
         print *, "read negation";  call parse_node_write (pn)
      end if
      pn_arg => parse_node_get_sub_ptr (pn, 2)
      call eval_node_compile_lvalue (en1, pn_arg, var_list)
      allocate (en)
      if (en1%type == EN_CONSTANT) then
         call eval_node_init_log (en, not_l (en1))
         call eval_node_final_rec (en1)
         deallocate (en1)
      else
         call eval_node_init_branch (en, var_str ("not"), V_LOG, en1)
         call eval_node_set_op1_log (en, not_l)
      end if
      if (debug) then
         call eval_node_write (en)
         print *, "done negation"
      end if
    end subroutine eval_node_compile_negation
```

Comparisons

Up to the loop, this is easy. There is always at least one comparison. This is evaluated, and the result is the logical node **en**. If it is constant, we keep its second sub-node as **en2**. (Thus, at the very end **en2** has to be deleted if **en** is (still) constant.)

If there is another comparison, we first check if the first comparison was constant. In that case, there are two possibilities: (i) it was true. Then, its right-hand side is compared with the new right-hand side, and the result replaces the previous one which is deleted. (ii) it was false. In this case, the result of the whole comparison is false, and we can exit the loop without evaluating anything else.

Now assume that the first comparison results in a valid branch, its second sub-node kept as **en2**. We first need a copy of this, which becomes the new left-hand side. If **en2** is constant, we make an identical constant node **en1**. Otherwise, we make **en1** an appropriate pointer node. Next, the first branch is saved as **en0** and we evaluate the comparison between **en1** and the a right-hand

403

side. If this turns out to be constant, there are again two possibilities: (i) true, then we revert to the previous result. (ii) false, then the wh

⟨*Expressions: procedures*⟩+≡

```
    recursive subroutine eval_node_compile_compared_expr (en, pn, var_list, type)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      integer, intent(in) :: type
      type(parse_node_t), pointer :: pn_comparison, pn_expr1
      type(eval_node_t), pointer :: en0, en1, en2
      if (debug) then
         print *, "read comparison";  call parse_node_write (pn)
      end if
      select case (type)
      case (V_INT, V_REAL)
         pn_expr1 => parse_node_get_sub_ptr (pn, tag="expr")
         call eval_node_compile_expr (en1, pn_expr1, var_list)
         pn_comparison => parse_node_get_next_ptr (pn_expr1, tag="comparison")
      case (V_STR)
         pn_expr1 => parse_node_get_sub_ptr (pn, tag="sexpr")
         call eval_node_compile_sexpr (en1, pn_expr1, var_list)
         pn_comparison => parse_node_get_next_ptr (pn_expr1, tag="str_comparison")
      end select
      call eval_node_compile_comparison &
           (en, en1, en2, pn_comparison, var_list, type)
      pn_comparison => parse_node_get_next_ptr (pn_comparison)
      SCAN_FURTHER: do while (associated (pn_comparison))
         if (en%type == EN_CONSTANT) then
            if (en%lval) then
               en1 => en2
               call eval_node_final_rec (en);  deallocate (en)
               call eval_node_compile_comparison &
                    (en, en1, en2, pn_comparison, var_list, type)
            else
               exit SCAN_FURTHER
            end if
         else
            allocate (en1)
            if (en2%type == EN_CONSTANT) then
               select case (en2%result_type)
               case (V_INT);  call eval_node_init_int    (en1, en2%ival)
               case (V_REAL); call eval_node_init_real   (en1, en2%rval)
               case (V_STR);  call eval_node_init_string (en1, en2%sval)
               end select
            else
               select case (en2%result_type)
               case (V_INT);  call eval_node_init_int_ptr &
                    (en1, var_str ("(previous)"), en2%ival, en2%value_is_known)
               case (V_REAL); call eval_node_init_real_ptr &
                    (en1, var_str ("(previous)"), en2%rval, en2%value_is_known)
               case (V_STR);  call eval_node_init_string_ptr &
                    (en1, var_str ("(previous)"), en2%sval, en2%value_is_known)
               end select
            end if
```

```
            en0 => en
            call eval_node_compile_comparison &
                (en, en1, en2, pn_comparison, var_list, type)
            if (en%type == EN_CONSTANT) then
                if (en%lval) then
                    call eval_node_final_rec (en);  deallocate (en)
                    en => en0
                else
                    call eval_node_final_rec (en0);  deallocate (en0)
                    exit SCAN_FURTHER
                end if
            else
                en1 => en
                allocate (en)
                call eval_node_init_branch (en, var_str ("and"), V_LOG, en0, en1)
                call eval_node_set_op2_log (en, and_ll)
            end if
        end if
        pn_comparison => parse_node_get_next_ptr (pn_comparison)
    end do SCAN_FURTHER
    if (en%type == EN_CONSTANT .and. associated (en2)) then
        call eval_node_final_rec (en2);  deallocate (en2)
    end if
    if (debug) then
        call eval_node_write (en)
        print *, "done compared_expr"
    end if
  end subroutine eval_node_compile_compared_expr
```

This takes two extra arguments: **en1**, the left-hand-side of the comparison, is already allocated and evaluated. **en2** (the right-hand side) and **en** (the result) are allocated by the routine. **pn** is the parse node which contains the operator and the right-hand side as subnodes.

If the result of the comparison is constant, **en1** is deleted but **en2** is kept, because it may be used in a subsequent comparison. **en** then becomes a constant. If the result is variable, **en** becomes a branch node which refers to **en1** and **en2**.

⟨*Expressions: procedures*⟩+≡

```
    recursive subroutine eval_node_compile_comparison &
        (en, en1, en2, pn, var_list, type)
    type(eval_node_t), pointer :: en, en1, en2
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    integer, intent(in) :: type
    type(parse_node_t), pointer :: pn_op, pn_arg
    type(string_t) :: key
    integer :: t1, t2
    type(var_entry_t), pointer :: var
    pn_op => parse_node_get_sub_ptr (pn)
    key = parse_node_get_key (pn_op)
    select case (type)
    case (V_INT, V_REAL)
        pn_arg => parse_node_get_next_ptr (pn_op, tag="expr")
        call eval_node_compile_expr (en2, pn_arg, var_list)
```

```
case (V_STR)
   pn_arg => parse_node_get_next_ptr (pn_op, tag="sexpr")
   call eval_node_compile_sexpr (en2, pn_arg, var_list)
end select
t1 = en1%result_type
t2 = en2%result_type
allocate (en)
if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
   select case (char (key))
   case ("<")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_lt_ii (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_lt_ir (en1, en2))
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_lt_ri (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_lt_rr (en1, en2))
         end select
      end select
   case (">")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_gt_ii (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_gt_ir (en1, en2))
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_gt_ri (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_gt_rr (en1, en2))
         end select
      end select
   case ("<=")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_le_ii (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_le_ir (en1, en2))
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_le_ri (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_le_rr (en1, en2))
         end select
      end select
   case (">=")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_ge_ii (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_ge_ir (en1, en2))
```

```
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_ge_ri (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_ge_rr (en1, en2))
         end select
      end select
   case ("==")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_eq_ii (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_eq_ir (en1, en2))
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_eq_ri (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_eq_rr (en1, en2))
         end select
      case (V_STR)
         select case (t2)
         case (V_STR);  call eval_node_init_log (en, comp_eq_ss (en1, en2))
         end select
      end select
   case ("<>")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_ne_ii (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_ne_ir (en1, en2))
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_ne_ri (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_ne_rr (en1, en2))
         end select
      case (V_STR)
         select case (t2)
         case (V_STR);  call eval_node_init_log (en, comp_ne_ss (en1, en2))
         end select
      end select
   case ("==~")
      var => var_list_get_var_ptr (var_list, var_str ("tolerance"))
      en1%tolerance => var_entry_get_rval_ptr (var)
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_sim_ii (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_sim_ir (en1, en2))
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_sim_ri (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_sim_rr (en1, en2))
```

```
                  end select
            case (V_STR)
               select case (t2)
               case (V_STR);  call eval_node_init_log (en, comp_eq_ss (en1, en2))
               end select
            end select
      case ("<>~")
         var => var_list_get_var_ptr (var_list, var_str ("tolerance"))
         en1%tolerance => var_entry_get_rval_ptr (var)
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT)
               call eval_node_init_log (en, comp_nsim_ii (en1, en2))
            case (V_REAL)
               call eval_node_init_log (en, comp_nsim_ir (en1, en2))
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT)
               call eval_node_init_log (en, comp_nsim_ri (en1, en2))
            case (V_REAL)
               call eval_node_init_log (en, comp_nsim_rr(en1, en2))
            end select
         case (V_STR)
            select case (t2)
            case (V_STR);  call eval_node_init_log (en, comp_ne_ss (en1, en2))
            end select
         end select
      end select
      call eval_node_final_rec (en1)
      deallocate (en1)
   else
      call eval_node_init_branch (en, key, V_LOG, en1, en2)
      select case (char (key))
      case ("<")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_log (en, comp_lt_ii)
            case (V_REAL); call eval_node_set_op2_log (en, comp_lt_ir)
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_log (en, comp_lt_ri)
            case (V_REAL); call eval_node_set_op2_log (en, comp_lt_rr)
            end select
         end select
      case (">")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_log (en, comp_gt_ii)
            case (V_REAL); call eval_node_set_op2_log (en, comp_gt_ir)
```

```
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_log (en, comp_gt_ri)
            case (V_REAL); call eval_node_set_op2_log (en, comp_gt_rr)
            end select
         end select
      case ("<=")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_log (en, comp_le_ii)
            case (V_REAL); call eval_node_set_op2_log (en, comp_le_ir)
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_log (en, comp_le_ri)
            case (V_REAL); call eval_node_set_op2_log (en, comp_le_rr)
            end select
         end select
      case (">=")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_log (en, comp_ge_ii)
            case (V_REAL); call eval_node_set_op2_log (en, comp_ge_ir)
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_log (en, comp_ge_ri)
            case (V_REAL); call eval_node_set_op2_log (en, comp_ge_rr)
            end select
         end select
      case ("==")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_log (en, comp_eq_ii)
            case (V_REAL); call eval_node_set_op2_log (en, comp_eq_ir)
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_log (en, comp_eq_ri)
            case (V_REAL); call eval_node_set_op2_log (en, comp_eq_rr)
            end select
         case (V_STR)
            select case (t2)
            case (V_STR);  call eval_node_set_op2_log (en, comp_eq_ss)
            end select
         end select
      case ("<>")
         select case (t1)
         case (V_INT)
            select case (t2)
```

```
                  case (V_INT);  call eval_node_set_op2_log (en, comp_ne_ii)
                  case (V_REAL); call eval_node_set_op2_log (en, comp_ne_ir)
                  end select
               case (V_REAL)
                  select case (t2)
                  case (V_INT);  call eval_node_set_op2_log (en, comp_ne_ri)
                  case (V_REAL); call eval_node_set_op2_log (en, comp_ne_rr)
                  end select
               case (V_STR)
                  select case (t2)
                  case (V_STR);  call eval_node_set_op2_log (en, comp_ne_ss)
                  end select
               end select
            case ("==~")
               select case (t1)
               case (V_INT)
                  select case (t2)
                  case (V_INT);  call eval_node_set_op2_log (en, comp_sim_ii)
                  case (V_REAL); call eval_node_set_op2_log (en, comp_sim_ir)
                  end select
               case (V_REAL)
                  select case (t2)
                  case (V_INT);  call eval_node_set_op2_log (en, comp_sim_ri)
                  case (V_REAL); call eval_node_set_op2_log (en, comp_sim_rr)
                  end select
               case (V_STR)
                  select case (t2)
                  case (V_STR);  call eval_node_set_op2_log (en, comp_eq_ss)
                  end select
               end select
               var => var_list_get_var_ptr (var_list, var_str ("tolerance"))
               en1%tolerance => var_entry_get_rval_ptr (var)
            case ("<>~")
               select case (t1)
               case (V_INT)
                  select case (t2)
                  case (V_INT);  call eval_node_set_op2_log (en, comp_nsim_ii)
                  case (V_REAL); call eval_node_set_op2_log (en, comp_nsim_ir)
                  end select
               case (V_REAL)
                  select case (t2)
                  case (V_INT);  call eval_node_set_op2_log (en, comp_nsim_ri)
                  case (V_REAL); call eval_node_set_op2_log (en, comp_nsim_rr)
                  end select
               case (V_STR)
                  select case (t2)
                  case (V_STR);  call eval_node_set_op2_log (en, comp_ne_ss)
                  end select
               end select
               var => var_list_get_var_ptr (var_list, var_str ("tolerance"))
               en1%tolerance => var_entry_get_rval_ptr (var)
            end select
       end if
     end subroutine eval_node_compile_comparison
```

Recording analysis data

The `record` command is actually a logical expression which always evaluates true.

⟨*Expressions: procedures*⟩+≡

```
recursive subroutine eval_node_compile_record_cmd (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_key, pn_tag, pn_arg, pn_arg1, pn_arg2
  type(eval_node_t), pointer :: en0, en1, en2
  type(var_entry_t), pointer :: var
  real(default), pointer :: event_weight
  if (debug) then
     print *, "read record_cmd";  call parse_node_write (pn)
  end if
  pn_key => parse_node_get_sub_ptr (pn)
  pn_tag => parse_node_get_next_ptr (pn_key)
  pn_arg => parse_node_get_next_ptr (pn_tag)
  select case (char (parse_node_get_key (pn_key)))
  case ("record")
     var => var_list_get_var_ptr (var_list, var_str ("event_weight"))
     if (associated (var)) then
        event_weight => var_entry_get_rval_ptr (var)
     else
        event_weight => null ()
     end if
  case ("record_unweighted")
     event_weight => null ()
  end select
  select case (char (parse_node_get_rule_key (pn_tag)))
  case ("analysis_id")
     allocate (en0)
     call eval_node_init_string (en0, parse_node_get_string (pn_tag))
  case default
     call eval_node_compile_sexpr (en0, pn_tag, var_list)
  end select
  allocate (en)
  if (associated (pn_arg)) then
     pn_arg1 => parse_node_get_sub_ptr (pn_arg)
     call eval_node_compile_expr (en1, pn_arg1, var_list)
     if (en1%result_type == V_INT) &
          call insert_conversion_node (en1, V_REAL)
     pn_arg2 => parse_node_get_next_ptr (pn_arg1)
     if (associated (pn_arg2)) then
        call eval_node_compile_expr (en2, pn_arg2, var_list)
        if (en2%result_type == V_INT) &
             call insert_conversion_node (en2, V_REAL)
        call eval_node_init_record_cmd (en, event_weight, en0, en1, en2)
     else
        call eval_node_init_record_cmd (en, event_weight, en0, en1)
```

```
         end if
      else
         call eval_node_init_record_cmd (en, event_weight, en0)
      end if
      if (debug) then
         call eval_node_write (en)
         print *, "done record_cmd"
      end if
   end subroutine eval_node_compile_record_cmd
```

Particle-list expressions

A particle expression is a particle list or a combination of particle-list value.

⟨*Expressions: procedures*⟩+≡
```
   recursive subroutine eval_node_compile_pexpr (en, pn, var_list)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     type(parse_node_t), pointer :: pn_pvalue, pn_combination, pn_op, pn_arg
     type(eval_node_t), pointer :: en1, en2
     type(prt_list_t) :: prt_list
     if (debug) then
        print *, "read pexpr";  call parse_node_write (pn)
     end if
     pn_pvalue => parse_node_get_sub_ptr (pn)
     call eval_node_compile_pvalue (en, pn_pvalue, var_list)
     pn_combination => parse_node_get_next_ptr (pn_pvalue, tag="combination")
     do while (associated (pn_combination))
        pn_op => parse_node_get_sub_ptr (pn_combination)
        pn_arg => parse_node_get_next_ptr (pn_op)
        en1 => en
        call eval_node_compile_pvalue (en2, pn_arg, var_list)
        allocate (en)
        if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
           call prt_list_combine (prt_list, en1%pval, en2%pval)
           call eval_node_init_prt_list (en, prt_list)
           call eval_node_final_rec (en1)
           call eval_node_final_rec (en2)
           deallocate (en1, en2)
        else
           call eval_node_init_branch &
                (en, var_str ("combine"), V_PTL, en1, en2)
           call eval_node_set_op2_ptl (en, combine_pp)
        end if
        pn_combination => parse_node_get_next_ptr (pn_combination)
     end do
     if (debug) then
        call eval_node_write (en)
        print *, "done pexpr"
     end if
   end subroutine eval_node_compile_pexpr
```

A particle-list value is a PDG-code array, a particle identifier, a variable, a (grouped) pexpr, a block pexpr, a conditional, or a particle-list function.

The `cexpr` node is responsible for transforming a constant PDG-code array into a particle list. It takes the code array as its first argument, the event particle list as its second argument, and the requested particle type (incoming/outgoing) as its zero-th argument. The result is the list of particle in the event that match the code array.

⟨*Expressions: procedures*⟩+≡

```
recursive subroutine eval_node_compile_pvalue (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_prefix_cexpr
  type(eval_node_t), pointer :: en1, en2, en0
  type(string_t) :: key
  type(var_entry_t), pointer :: var
  logical, save, target :: known = .true.
  if (debug) then
     print *, "read pvalue";  call parse_node_write (pn)
  end if
  select case (char (parse_node_get_rule_key (pn)))
  case ("pexpr_src")
     call eval_node_compile_prefix_cexpr (en1, pn, var_list)
     allocate (en2)
     var => var_list_get_var_ptr (var_list, var_str ("@evt"))
     if (associated (var)) then
        call eval_node_init_prt_list_ptr &
             (en2, var_str ("@evt"), var_entry_get_pval_ptr (var), known)
        allocate (en)
        call eval_node_init_branch &
             (en, var_str ("prt_selection"), V_PTL, en1, en2)
        call eval_node_set_op2_ptl (en, select_pdg_ca)
        allocate (en0)
        pn_prefix_cexpr => parse_node_get_sub_ptr (pn)
        key = parse_node_get_rule_key (pn_prefix_cexpr)
        select case (char (key))
        case ("incoming_prt")
           call eval_node_init_int (en0, PRT_INCOMING)
           en%arg0 => en0
        case ("outgoing_prt")
           call eval_node_init_int (en0, PRT_OUTGOING)
           en%arg0 => en0
        end select
     else
        call parse_node_write (pn)
        call msg_bug (" Missing event data while compiling pvalue")
     end if
  case ("pvariable")
     call eval_node_compile_variable (en, pn, var_list, V_PTL)
  case ("pexpr")
     call eval_node_compile_pexpr (en, pn, var_list)
  case ("block_pexpr")
     call eval_node_compile_block_expr (en, pn, var_list, V_PTL)
```

```
      case ("conditional_pexpr")
         call eval_node_compile_conditional (en, pn, var_list, V_PTL)
      case ("join_fun", "combine_fun", "collect_fun", "select_fun", &
            "extract_fun", "sort_fun")
         call eval_node_compile_prt_function (en, pn, var_list)
      case default
         call parse_node_mismatch &
              ("prefix_cexpr|pvariable|" // &
               "grouped_pexpr|block_pexpr|conditional_pexpr|" // &
               "prt_function", pn)
      end select
      if (debug) then
         call eval_node_write (en)
         print *, "done pvalue"
      end if
   end subroutine eval_node_compile_pvalue
```

Particle functions

This combines the treatment of 'join', 'combine', 'collect', 'select', and 'extract'
which all have the same syntax. The one or two argument nodes are allocated.
If there is a condition, the condition node is also allocated as a logical expres-
sion, for which the variable list is augmented by the appropriate (unary/binary)
observables.

⟨*Expressions: procedures*⟩+≡
```
   recursive subroutine eval_node_compile_prt_function (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_clause, pn_key, pn_cond, pn_args
      type(parse_node_t), pointer :: pn_arg0, pn_arg1, pn_arg2
      type(eval_node_t), pointer :: en0, en1, en2
      type(string_t) :: key
      if (debug) then
         print *, "read prt_function";  call parse_node_write (pn)
      end if
      pn_clause => parse_node_get_sub_ptr (pn)
      pn_key  => parse_node_get_sub_ptr (pn_clause)
      pn_cond => parse_node_get_next_ptr (pn_key)
      if (associated (pn_cond)) &
           pn_arg0 => parse_node_get_sub_ptr (pn_cond, 2)
      pn_args => parse_node_get_next_ptr (pn_clause)
      pn_arg1 => parse_node_get_sub_ptr (pn_args)
      pn_arg2 => parse_node_get_next_ptr (pn_arg1)
      key = parse_node_get_key (pn_key)
      call eval_node_compile_pexpr (en1, pn_arg1, var_list)
      allocate (en)
      if (.not. associated (pn_arg2)) then
         select case (char (key))
         case ("collect")
            call eval_node_init_prt_fun_unary (en, en1, key, collect_p)
         case ("select")
```

```
            call eval_node_init_prt_fun_unary (en, en1, key, select_p)
         case ("extract")
            call eval_node_init_prt_fun_unary (en, en1, key, extract_p)
         case ("sort")
            call eval_node_init_prt_fun_unary (en, en1, key, sort_p)
         case default
            call msg_bug (" Unary particle function '" // char (key) // &
                 "' undefined")
         end select
      else
         call eval_node_compile_pexpr (en2, pn_arg2, var_list)
         select case (char (key))
         case ("join")
            call eval_node_init_prt_fun_binary (en, en1, en2, key, join_pp)
         case ("combine")
            call eval_node_init_prt_fun_binary (en, en1, en2, key, combine_pp)
         case ("collect")
            call eval_node_init_prt_fun_binary (en, en1, en2, key, collect_pp)
         case ("select")
            call eval_node_init_prt_fun_binary (en, en1, en2, key, select_pp)
         case ("sort")
            call eval_node_init_prt_fun_binary (en, en1, en2, key, sort_pp)
         case default
            call msg_bug (" Binary particle function '" // char (key) // &
                 "' undefined")
         end select
      end if
      if (associated (pn_cond)) then
         call eval_node_set_observables (en, var_list)
         select case (char (key))
         case ("extract", "sort")
            call eval_node_compile_expr (en0, pn_arg0, en%var_list)
         case default
            call eval_node_compile_lexpr (en0, pn_arg0, en%var_list)
         end select
         en%arg0 => en0
      end if
      if (debug) then
         call eval_node_write (en)
         print *, "done prt_function"
      end if
   end subroutine eval_node_compile_prt_function
```

The `eval` expression is similar, but here the expression `arg0` is mandatory, and
the whole thing evaluates to a numeric value.

⟨*Expressions: procedures*⟩+≡

```
   recursive subroutine eval_node_compile_eval_function (en, pn, var_list)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     type(parse_node_t), pointer :: pn_key, pn_arg0, pn_args, pn_arg1, pn_arg2
     type(eval_node_t), pointer :: en0, en1, en2
     type(string_t) :: key
```

```
      if (debug) then
         print *, "read eval_function";  call parse_node_write (pn)
      end if
      pn_key => parse_node_get_sub_ptr (pn)
      pn_arg0 => parse_node_get_next_ptr (pn_key)
      pn_args => parse_node_get_next_ptr (pn_arg0)
      pn_arg1 => parse_node_get_sub_ptr (pn_args)
      pn_arg2 => parse_node_get_next_ptr (pn_arg1)
      key = parse_node_get_key (pn_key)
      call eval_node_compile_pexpr (en1, pn_arg1, var_list)
      allocate (en)
      if (.not. associated (pn_arg2)) then
         call eval_node_init_eval_fun_unary (en, en1, key)
      else
         call eval_node_compile_pexpr (en2, pn_arg2, var_list)
         call eval_node_init_eval_fun_binary (en, en1, en2, key)
      end if
      call eval_node_set_observables (en, var_list)
      call eval_node_compile_expr (en0, pn_arg0, en%var_list)
      if (en0%result_type /= V_REAL) &
           call msg_fatal (" 'eval' function does not result in real value")
      call eval_node_set_expr (en, en0)
      if (debug) then
         call eval_node_write (en)
         print *, "done eval_function"
      end if
    end subroutine eval_node_compile_eval_function
```

Logical functions of particle lists.

⟨*Expressions: procedures*⟩+≡

```
    recursive subroutine eval_node_compile_log_function (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_key, pn_arg0, pn_args, pn_arg1, pn_arg2
      type(eval_node_t), pointer :: en0, en1, en2
      type(string_t) :: key
      if (debug) then
         print *, "read log_function";  call parse_node_write (pn)
      end if
      pn_key => parse_node_get_sub_ptr (pn)
      pn_arg0 => parse_node_get_next_ptr (pn_key)
      pn_args => parse_node_get_next_ptr (pn_arg0)
      pn_arg1 => parse_node_get_sub_ptr (pn_args)
      pn_arg2 => parse_node_get_next_ptr (pn_arg1)
      key = parse_node_get_key (pn_key)
      call eval_node_compile_pexpr (en1, pn_arg1, var_list)
      allocate (en)
      if (.not. associated (pn_arg2)) then
         select case (char (key))
         case ("all")
            call eval_node_init_log_fun_unary (en, en1, key, all_p)
         case ("any")
            call eval_node_init_log_fun_unary (en, en1, key, any_p)
```

```
      case ("no")
         call eval_node_init_log_fun_unary (en, en1, key, no_p)
      end select
   else
      call eval_node_compile_pexpr (en2, pn_arg2, var_list)
      select case (char (key))
      case ("all")
         call eval_node_init_log_fun_binary (en, en1, en2, key, all_pp)
      case ("any")
         call eval_node_init_log_fun_binary (en, en1, en2, key, any_pp)
      case ("no")
         call eval_node_init_log_fun_binary (en, en1, en2, key, no_pp)
      end select
   end if
   call eval_node_set_observables (en, var_list)
   call eval_node_compile_lexpr (en0, pn_arg0, en%var_list)
   call eval_node_set_expr (en, en0)
   if (debug) then
      call eval_node_write (en)
      print *, "done log_function"
   end if
end subroutine eval_node_compile_log_function
```

Integer functions of particle lists.

⟨*Expressions: procedures*⟩+≡

```
recursive subroutine eval_node_compile_int_function (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_clause, pn_key, pn_cond, pn_args
  type(parse_node_t), pointer :: pn_arg0, pn_arg1, pn_arg2
  type(eval_node_t), pointer :: en0, en1, en2
  type(string_t) :: key
  if (debug) then
     print *, "read int_function";  call parse_node_write (pn)
  end if
  pn_clause => parse_node_get_sub_ptr (pn)
  pn_key  => parse_node_get_sub_ptr (pn_clause)
  pn_cond => parse_node_get_next_ptr (pn_key)
  if (associated (pn_cond)) &
       pn_arg0 => parse_node_get_sub_ptr (pn_cond, 2)
  pn_args => parse_node_get_next_ptr (pn_clause)
  pn_arg1 => parse_node_get_sub_ptr (pn_args)
  pn_arg2 => parse_node_get_next_ptr (pn_arg1)
  key = parse_node_get_key (pn_key)
  call eval_node_compile_pexpr (en1, pn_arg1, var_list)
  allocate (en)
  if (.not. associated (pn_arg2)) then
     select case (char (key))
     case ("count")
        call eval_node_init_int_fun_unary (en, en1, key, count_a)
     end select
  else
     call eval_node_compile_pexpr (en2, pn_arg2, var_list)
```

```
      select case (char (key))
      case ("count")
         call eval_node_init_int_fun_binary (en, en1, en2, key, count_pp)
      end select
   end if
   if (associated (pn_cond)) then
      call eval_node_set_observables (en, var_list)
      call eval_node_compile_lexpr (en0, pn_arg0, en%var_list)
      call eval_node_set_expr (en, en0, V_INT)
   end if
   if (debug) then
      call eval_node_write (en)
      print *, "done int_function"
   end if
end subroutine eval_node_compile_int_function
```

PDG-code arrays

A PDG-code expression is either prefixed by `incoming` or `outgoing`, a block, or a conditional. In any case, it evaluates to a constant.

⟨*Expressions: procedures*⟩+≡
```
recursive subroutine eval_node_compile_prefix_cexpr (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_avalue, pn_prt
  type(string_t) :: key
  if (debug) then
     print *, "read prefix_cexpr";  call parse_node_write (pn)
  end if
  pn_avalue => parse_node_get_sub_ptr (pn)
  key = parse_node_get_rule_key (pn_avalue)
  select case (char (key))
  case ("incoming_prt")
     pn_prt => parse_node_get_sub_ptr (pn_avalue, 2)
     call eval_node_compile_cexpr (en, pn_prt, var_list)
  case ("outgoing_prt")
     pn_prt => parse_node_get_sub_ptr (pn_avalue, 1)
     call eval_node_compile_cexpr (en, pn_prt, var_list)
  case default
     call parse_node_mismatch &
          ("incoming_prt|outgoing_prt", &
           pn_avalue)
  end select
  if (debug) then
     call eval_node_write (en)
     print *, "done prefix_cexpr"
  end if
end subroutine eval_node_compile_prefix_cexpr
```

A PDG array is a string of PDG code definitions (or aliases), concatenated by ':'. The code definitions may be variables which are not defined at compile time,

so we have to allocate sub-nodes. This analogous to `eval_node_compile_term`.

⟨*Expressions: procedures*⟩+≡

```
recursive subroutine eval_node_compile_cexpr (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_prt, pn_concatenation
  type(eval_node_t), pointer :: en1, en2
  type(pdg_array_t) :: aval
  if (debug) then
     print *, "read cexpr";  call parse_node_write (pn)
  end if
  pn_prt => parse_node_get_sub_ptr (pn)
  call eval_node_compile_avalue (en, pn_prt, var_list)
  pn_concatenation => parse_node_get_next_ptr (pn_prt)
  do while (associated (pn_concatenation))
     pn_prt => parse_node_get_sub_ptr (pn_concatenation, 2)
     en1 => en
     call eval_node_compile_avalue (en2, pn_prt, var_list)
     allocate (en)
     if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
        call concat_cc (aval, en1, en2)
        call eval_node_init_pdg_array (en, aval)
        call eval_node_final_rec (en1)
        call eval_node_final_rec (en2)
        deallocate (en1, en2)
     else
        call eval_node_init_branch (en, var_str (":"), V_PDG, en1, en2)
        call eval_node_set_op2_pdg (en, concat_cc)
     end if
     pn_concatenation => parse_node_get_next_ptr (pn_concatenation)
  end do
  if (debug) then
     call eval_node_write (en)
     print *, "done cexpr"
  end if
end subroutine eval_node_compile_cexpr
```

Compile a PDG-code type value. It may be either an integer expression or a variable of type PDG array, optionally quoted.

⟨*Expressions: procedures*⟩+≡

```
recursive subroutine eval_node_compile_avalue (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  if (debug) then
     print *, "read avalue";  call parse_node_write (pn)
  end if
  select case (char (parse_node_get_rule_key (pn)))
  case ("pdg_code")
     call eval_node_compile_pdg_code (en, pn, var_list)
  case ("cvariable", "variable", "prt_name")
     call eval_node_compile_cvariable (en, pn, var_list)
```

```
      case ("cexpr")
         call eval_node_compile_cexpr (en, pn, var_list)
      case ("block_cexpr")
         call eval_node_compile_block_expr (en, pn, var_list, V_PDG)
      case ("conditional_cexpr")
         call eval_node_compile_conditional (en, pn, var_list, V_PDG)
      case default
         call parse_node_mismatch &
              ("grouped_cexpr|block_cexpr|conditional_cexpr|" // &
               "pdg_code|cvariable|prt_name", pn)
      end select
      if (debug) then
         call eval_node_write (en)
         print *, "done avalue"
      end if
    end subroutine eval_node_compile_avalue
```

Compile a PDG-code expression, which is the key PDG with an integer expression
as argument. The procedure is analogous to `eval_node_compile_unary_function`.

⟨*Expressions: procedures*⟩+≡

```
    subroutine eval_node_compile_pdg_code (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in), target :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_arg
      type(eval_node_t), pointer :: en1
      type(string_t) :: key
      type(pdg_array_t) :: aval
      integer :: t
      if (debug) then
         print *, "read PDG code";  call parse_node_write (pn)
      end if
      pn_arg => parse_node_get_sub_ptr (pn, 2)
      call eval_node_compile_expr &
           (en1, parse_node_get_sub_ptr (pn_arg, tag="expr"), var_list)
      t = en1%result_type
      allocate (en)
      key = "PDG"
      if (en1%type == EN_CONSTANT) then
         select case (t)
         case (V_INT)
            call pdg_i (aval, en1)
            call eval_node_init_pdg_array (en, aval)
         case default;  call eval_type_error (pn, char (key), t)
         end select
         call eval_node_final_rec (en1)
         deallocate (en1)
      else
         select case (t)
         case (V_INT);  call eval_node_set_op1_pdg (en, pdg_i)
         case default;  call eval_type_error (pn, char (key), t)
         end select
      end if
```

```
      if (debug) then
         call eval_node_write (en)
         print *, "done function"
      end if
   end subroutine eval_node_compile_pdg_code
```

This is entirely analogous to `eval_node_compile_variable`. However, PDG-array variables occur in different contexts.

To avoid name clashes between PDG-array variables and ordinary variables, we prepend a character (∗). This is not visible to the user.

⟨*Expressions: procedures*⟩+≡
```
   subroutine eval_node_compile_cvariable (en, pn, var_list)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in), target :: pn
     type(var_list_t), intent(in), target :: var_list
     type(parse_node_t), pointer :: pn_name
     type(string_t) :: var_name
     type(var_entry_t), pointer :: var
     type(pdg_array_t), target, save :: no_aval
     logical, target, save :: unknown = .false.
     if (debug) then
        print *, "read cvariable";  call parse_node_write (pn)
     end if
   !    select case (char (parse_node_get_rule_key (pn)))
   !    case ("cvariable");  pn_name => parse_node_get_sub_ptr (pn, 2)
   !    case default;        pn_name => pn
   !    end select
     pn_name => pn
     var_name = parse_node_get_string (pn_name)
     var => var_list_get_var_ptr (var_list, var_name, V_PDG)
     allocate (en)
     if (associated (var)) then
        call eval_node_init_pdg_array_ptr &
              (en, var_entry_get_name (var), var_entry_get_aval_ptr (var), &
               var_entry_get_known_ptr (var))
     else
        call parse_node_write (pn)
        call msg_error ("This PDG-array variable is undefined at this point")
        call eval_node_init_pdg_array_ptr (en, var_name, no_aval, unknown)
     end if
     if (debug) then
        call eval_node_write (en)
        print *, "done cvariable"
     end if
   end subroutine eval_node_compile_cvariable
```

String expressions

A string expression is either a string value or a concatenation of string values.

⟨*Expressions: procedures*⟩+≡
```
   recursive subroutine eval_node_compile_sexpr (en, pn, var_list)
     type(eval_node_t), pointer :: en
```

```
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_svalue, pn_concatenation, pn_op, pn_arg
    type(eval_node_t), pointer :: en1, en2
    type(string_t) :: string
    if (debug) then
       print *, "read sexpr";  call parse_node_write (pn)
    end if
    pn_svalue => parse_node_get_sub_ptr (pn)
    call eval_node_compile_svalue (en, pn_svalue, var_list)
    pn_concatenation => &
         parse_node_get_next_ptr (pn_svalue, tag="str_concatenation")
    do while (associated (pn_concatenation))
       pn_op => parse_node_get_sub_ptr (pn_concatenation)
       pn_arg => parse_node_get_next_ptr (pn_op)
       en1 => en
       call eval_node_compile_svalue (en2, pn_arg, var_list)
       allocate (en)
       if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
          call concat_ss (string, en1, en2)
          call eval_node_init_string (en, string)
          call eval_node_final_rec (en1)
          call eval_node_final_rec (en2)
          deallocate (en1, en2)
       else
          call eval_node_init_branch &
               (en, var_str ("concat"), V_STR, en1, en2)
          call eval_node_set_op2_str (en, concat_ss)
       end if
       pn_concatenation => parse_node_get_next_ptr (pn_concatenation)
    end do
    if (debug) then
       call eval_node_write (en)
       print *, "done sexpr"
    end if
  end subroutine eval_node_compile_sexpr
```

A string value is a string literal, a variable, a (grouped) sexpr, a block sexpr, or
a conditional.

⟨*Expressions: procedures*⟩+≡
```
    recursive subroutine eval_node_compile_svalue (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      if (debug) then
         print *, "read svalue";  call parse_node_write (pn)
      end if
      select case (char (parse_node_get_rule_key (pn)))
      case ("svariable")
         call eval_node_compile_variable (en, pn, var_list, V_STR)
      case ("sexpr")
         call eval_node_compile_sexpr (en, pn, var_list)
      case ("block_sexpr")
```

```
         call eval_node_compile_block_expr (en, pn, var_list, V_STR)
      case ("conditional_sexpr")
         call eval_node_compile_conditional (en, pn, var_list, V_STR)
      case ("sprintf_fun")
         call eval_node_compile_sprintf (en, pn, var_list)
      case ("sprintd_fun")
         call eval_node_compile_sprint (en, pn, var_list)
      case ("string_literal")
         allocate (en)
         call eval_node_init_string (en, parse_node_get_string (pn))
      case default
         call parse_node_mismatch &
              ("svariable|" // &
               "grouped_sexpr|block_sexpr|conditional_sexpr|" // &
               "string_function|string_literal", pn)
      end select
      if (debug) then
         call eval_node_write (en)
         print *, "done svalue"
      end if
    end subroutine eval_node_compile_svalue
```

There are currently one string function, `sprintf` and `sprint`. For `sprintf`,
The first argument (no brackets) is the format string, the optional arguments
in brackets are the expressions or variables to be formatted.

⟨*Expressions: procedures*⟩+≡

```
    recursive subroutine eval_node_compile_sprintf (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_clause, pn_key, pn_args
      type(parse_node_t), pointer :: pn_arg0
      type(eval_node_t), pointer :: en0, en1
      integer :: n_args
      type(string_t) :: key
      if (debug) then
         print *, "read sprintf_fun";  call parse_node_write (pn)
      end if
      pn_clause => parse_node_get_sub_ptr (pn)
      pn_key  => parse_node_get_sub_ptr (pn_clause)
      pn_arg0 => parse_node_get_next_ptr (pn_key)
      pn_args => parse_node_get_next_ptr (pn_clause)
      call eval_node_compile_sexpr (en0, pn_arg0, var_list)
      if (associated (pn_args)) then
         call eval_node_compile_sprintf_args (en1, pn_args, var_list, n_args)
      else
         n_args = 0
         en1 => null ()
      end if
      allocate (en)
      key = parse_node_get_key (pn_key)
      call eval_node_init_format_string (en, en0, en1, key, n_args)
      if (debug) then
```

423

```
      call eval_node_write (en)
      print *, "done sprintf_fun"
    end if
  end subroutine eval_node_compile_sprintf
```

sprint is similar, just the format string is missing.

⟨Expressions: procedures⟩+≡
```
    recursive subroutine eval_node_compile_sprint (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_clause, pn_key, pn_args
      type(eval_node_t), pointer :: en1
      integer :: n_args
      type(string_t) :: key
      if (debug) then
         print *, "read sprintd_fun";  call parse_node_write (pn)
      end if
      pn_key  => parse_node_get_sub_ptr (pn)
      pn_args => parse_node_get_next_ptr (pn_key)
      if (associated (pn_args)) then
         call eval_node_compile_sprintf_args (en1, pn_args, var_list, n_args)
      else
         n_args = 0
         en1 => null ()
      end if
      allocate (en)
      key = parse_node_get_key (pn_key)
      call eval_node_init_format_string (en, null (), en1, key, n_args)
      if (debug) then
         call eval_node_write (en)
         print *, "done sprintd_fun"
      end if
    end subroutine eval_node_compile_sprint
```

⟨Expressions: procedures⟩+≡
```
    subroutine eval_node_compile_sprintf_args (en, pn, var_list, n_args)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      integer, intent(out) :: n_args
      type(parse_node_t), pointer :: pn_arg
      integer :: i
      type(eval_node_t), pointer :: en1, en2
      n_args = parse_node_get_n_sub (pn)
      en => null ()
      do i = n_args, 1, -1
         pn_arg => parse_node_get_sub_ptr (pn, i)
         select case (char (parse_node_get_rule_key (pn_arg)))
         case ("lvariable")
            call eval_node_compile_variable (en1, pn_arg, var_list, V_LOG)
         case ("svariable")
            call eval_node_compile_variable (en1, pn_arg, var_list, V_STR)
```

```
        case ("expr")
           call eval_node_compile_expr (en1, pn_arg, var_list)
        case default
           call parse_node_mismatch ("variable|svariable|lvariable|expr", pn_arg)
        end select
        if (associated (en)) then
           en2 => en
           allocate (en)
           call eval_node_init_branch &
                (en, var_str ("sprintf_arg"), V_NONE, en1, en2)
        else
           allocate (en)
           call eval_node_init_branch &
                (en, var_str ("sprintf_arg"), V_NONE, en1)
        end if
     end do
  end subroutine eval_node_compile_sprintf_args
```

Evaluation. We allocate the argument list and apply the Fortran wrapper for the `sprintf` function. In the `sprint` case, build a format string appropriate for the argument types.

⟨*Expressions: procedures*⟩+≡
```
  subroutine evaluate_sprintf (string, n_args, en_fmt, en_arg)
    type(string_t), intent(out) :: string
    integer, intent(in) :: n_args
    type(eval_node_t), pointer :: en_fmt
    type(eval_node_t), intent(in), optional, target :: en_arg
    type(eval_node_t), pointer :: en_branch, en_var
    type(sprintf_arg_t), dimension(:), allocatable :: arg
    type(string_t) :: fmt
    integer :: i
    if (present (en_arg)) then
       allocate (arg (n_args))
       en_branch => en_arg
       do i = 1, n_args
          en_var => en_branch%arg1
          select case (en_var%result_type)
          case (V_LOG);  call sprintf_arg_init (arg(i), en_var%lval)
          case (V_INT);  call sprintf_arg_init (arg(i), en_var%ival)
          case (V_REAL); call sprintf_arg_init (arg(i), en_var%rval)
          case (V_STR);  call sprintf_arg_init (arg(i), en_var%sval)
          case default
             call eval_node_write (en_var)
             call msg_error ("sprintf is implemented " &
                  // "for logical, integer, real, and string values only")
          end select
          en_branch => en_branch%arg2
       end do
    else
       allocate (arg(0))
    end if
    if (associated (en_fmt)) then
       string = sprintf (en_fmt%sval, arg)
```

```
    else
       fmt = ""
       do i = 1, n_args
          if (i > 1)  fmt = fmt // " "
          select case (en_var%result_type)
          case (V_LOG);  fmt = fmt // "%s"
          case (V_INT);  fmt = fmt // "%i"
          case (V_REAL); fmt = fmt // "%g"
          case (V_STR);  fmt = fmt // "%s"
          end select
       end do
       string = sprintf (fmt, arg)
    end if
  end subroutine evaluate_sprintf
```

5.5.5 Auxiliary functions for the compiler

Issue an error that the current node could not be compiled because of type mismatch:

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_type_error (pn, string, t)
      type(parse_node_t), intent(in) :: pn
      character(*), intent(in) :: string
      integer, intent(in) :: t
      type(string_t) :: type
      select case (t)
      case (V_NONE); type = "(none)"
      case (V_LOG);  type = "'logical'"
      case (V_INT);  type = "'integer'"
      case (V_REAL); type = "'real'"
      case (V_CMPLX); type = "'complex'"
      case default;  type = "(unknown)"
      end select
      call parse_node_write (pn)
      call msg_fatal (" The " // string // &
           " operation is not defined for the given argument type " // &
           char (type))
    end subroutine eval_type_error
```

If two numerics are combined, the result is integer if both arguments are integer, if one is integer and the other real or both are real, than its argument is real, otherwise complex.

⟨*Expressions: procedures*⟩+≡
```
    function numeric_result_type (t1, t2) result (t)
      integer, intent(in) :: t1, t2
      integer :: t
      if (t1 == V_INT .and. t2 == V_INT) then
         t = V_INT
      else if (t1 == V_INT .and. t2 == V_REAL) then
         t = V_REAL
      else if (t1 == V_REAL .and. t2 == V_INT) then
         t = V_REAL
```

```
      else if (t1 == V_REAL .and. t2 == V_REAL) then
         t = V_REAL
      else
         t = V_CMPLX
      end if
  end function numeric_result_type
```

5.5.6 Evaluation

Evaluation is done recursively. For leaf nodes nothing is to be done.

Evaluating particle-list functions: First, we evaluate the particle lists. If a
condition is present, we assign the particle pointers of the condition node to the
allocated particle entries in the parent node, keeping in mind that the observ-
ables in the variable stack used for the evaluation of the condition also contain
pointers to these entries. Then, the assigned procedure is evaluated, which sets
the particle list in the parent node. If required, the procedure evaluates the
condition node once for each (pair of) particles to determine the result.

⟨*Expressions: procedures*⟩+≡
```
    recursive subroutine eval_node_evaluate (en)
      type(eval_node_t), intent(inout) :: en
      logical :: exist
      select case (en%type)
      case (EN_UNARY)
         if (associated (en%arg1)) then
            call eval_node_evaluate (en%arg1)
            en%value_is_known = en%arg1%value_is_known
         else
            en%value_is_known = .false.
         end if
         if (en%value_is_known) then
            select case (en%result_type)
            case (V_LOG);   en%lval = en% op1_log   (en%arg1)
            case (V_INT);   en%ival = en% op1_int   (en%arg1)
            case (V_REAL);  en%rval = en% op1_real  (en%arg1)
            case (V_CMPLX); en%cval = en% op1_cmplx (en%arg1)
            case (V_PDG);
                call en% op1_pdg  (en%aval, en%arg1)
            case (V_PTL)
               if (associated (en%arg0)) then
                  call en% op1_ptl (en%pval, en%arg1, en%arg0)
               else
                  call en% op1_ptl (en%pval, en%arg1)
               end if
            case (V_STR)
               call en% op1_str (en%sval, en%arg1)
            end select
         end if
      case (EN_BINARY)
         if (associated (en%arg1) .and. associated (en%arg2)) then
            call eval_node_evaluate (en%arg1)
            call eval_node_evaluate (en%arg2)
            en%value_is_known = &
```

```
                  en%arg1%value_is_known .and. en%arg2%value_is_known
          else
             en%value_is_known = .false.
          end if
          if (en%value_is_known) then
             select case (en%result_type)
             case (V_LOG);   en%lval = en% op2_log  (en%arg1, en%arg2)
             case (V_INT);   en%ival = en% op2_int  (en%arg1, en%arg2)
             case (V_REAL); en%rval = en% op2_real (en%arg1, en%arg2)
             case (V_CMPLX); en%cval = en% op2_cmplx (en%arg1, en%arg2)
             case (V_PDG)
                call en% op2_pdg  (en%aval, en%arg1, en%arg2)
             case (V_PTL)
                if (associated (en%arg0)) then
                   call en% op2_ptl (en%pval, en%arg1, en%arg2, en%arg0)
                else
                   call en% op2_ptl (en%pval, en%arg1, en%arg2)
                end if
             case (V_STR)
                call en% op2_str (en%sval, en%arg1, en%arg2)
             end select
          end if
       case (EN_BLOCK)
          if (associated (en%arg1) .and. associated (en%arg0)) then
             call eval_node_evaluate (en%arg1)
             call eval_node_evaluate (en%arg0)
             en%value_is_known = en%arg0%value_is_known
          else
             en%value_is_known = .false.
          end if
          if (en%value_is_known) then
             select case (en%result_type)
             case (V_LOG);   en%lval = en%arg0%lval
             case (V_INT);   en%ival = en%arg0%ival
             case (V_REAL);  en%rval = en%arg0%rval
             case (V_CMPLX); en%cval = en%arg0%cval
             case (V_PDG);   en%aval = en%arg0%aval
             case (V_PTL);   en%pval = en%arg0%pval
             case (V_STR);   en%sval = en%arg0%sval
             end select
          end if
       case (EN_CONDITIONAL)
          if (associated (en%arg0)) then
             call eval_node_evaluate (en%arg0)
             en%value_is_known = en%arg0%value_is_known
          else
             en%value_is_known = .false.
          end if
          if (en%arg0%value_is_known) then
             if (en%arg0%lval) then
                call eval_node_evaluate (en%arg1)
                en%value_is_known = en%arg1%value_is_known
                if (en%value_is_known) then
                   select case (en%result_type)
```

```
                case (V_LOG);   en%lval = en%arg1%lval
                case (V_INT);   en%ival = en%arg1%ival
                case (V_REAL);  en%rval = en%arg1%rval
                case (V_CMPLX); en%cval = en%arg1%cval
                case (V_PDG);   en%aval = en%arg1%aval
                case (V_PTL);   en%pval = en%arg1%pval
                case (V_STR);   en%sval = en%arg1%sval
                end select
             end if
          else
             call eval_node_evaluate (en%arg2)
             en%value_is_known = en%arg2%value_is_known
             if (en%value_is_known) then
                select case (en%result_type)
                case (V_LOG);   en%lval = en%arg2%lval
                case (V_INT);   en%ival = en%arg2%ival
                case (V_REAL);  en%rval = en%arg2%rval
                case (V_CMPLX); en%cval = en%arg2%cval
                case (V_PDG);   en%aval = en%arg2%aval
                case (V_PTL);   en%pval = en%arg2%pval
                case (V_STR);   en%sval = en%arg2%sval
                end select
             end if
          end if
       end if
    case (EN_RECORD_CMD)
       exist = .true.
       call eval_node_evaluate (en%arg0)
       if (en%arg0%value_is_known) then
          if (associated (en%arg1)) then
             call eval_node_evaluate (en%arg1)
             if (en%arg1%value_is_known) then
                if (associated (en%arg2)) then
                   call eval_node_evaluate (en%arg2)
                   if (en%arg2%value_is_known) then
                      if (associated (en%rval)) then
                         call analysis_record_data (en%arg0%sval, &
                              en%arg1%rval, en%arg2%rval, &
                              weight=en%rval, exist=exist)
                      else
                         call analysis_record_data (en%arg0%sval, &
                              en%arg1%rval, en%arg2%rval, exist=exist)
                      end if
                   end if
                else
                   if (associated (en%rval)) then
                      call analysis_record_data (en%arg0%sval, &
                           en%arg1%rval, weight=en%rval, exist=exist)
                   else
                      call analysis_record_data (en%arg0%sval, &
                           en%arg1%rval, exist=exist)
                   end if
                end if
             end if
          end if
```

```fortran
      else
         if (associated (en%rval)) then
            call analysis_record_data (en%arg0%sval, 1._default, &
                 weight=en%rval, exist=exist)
         else
            call analysis_record_data (en%arg0%sval, 1._default, &
                 exist=exist)
         end if
      end if
      if (.not. exist) then
         call msg_error ("Analysis object '" // char (en%arg0%sval) &
              // "' is undefined")
         en%arg0%value_is_known = .false.
      end if
   end if
case (EN_OBS1_INT)
   en%ival = en% obs1_int (en%prt1)
   en%value_is_known = .true.
case (EN_OBS2_INT)
   en%ival = en% obs2_int (en%prt1, en%prt2)
   en%value_is_known = .true.
case (EN_OBS1_REAL)
   en%rval = en% obs1_real (en%prt1)
   en%value_is_known = .true.
case (EN_OBS2_REAL)
   en%rval = en% obs2_real (en%prt1, en%prt2)
   en%value_is_known = .true.
case (EN_PRT_FUN_UNARY)
   call eval_node_evaluate (en%arg1)
   en%value_is_known = en%arg1%value_is_known
   if (en%value_is_known) then
      if (associated (en%arg0)) then
         en%arg0%index => en%index
         en%arg0%prt1 => en%prt1
         call en% op1_ptl (en%pval, en%arg1, en%arg0)
      else
         call en% op1_ptl (en%pval, en%arg1)
      end if
   end if
case (EN_PRT_FUN_BINARY)
   call eval_node_evaluate (en%arg1)
   call eval_node_evaluate (en%arg2)
   en%value_is_known = &
        en%arg1%value_is_known .and. en%arg2%value_is_known
   if (en%value_is_known) then
      if (associated (en%arg0)) then
         en%arg0%index => en%index
         en%arg0%prt1 => en%prt1
         en%arg0%prt2 => en%prt2
         call en% op2_ptl (en%pval, en%arg1, en%arg2, en%arg0)
      else
         call en% op2_ptl (en%pval, en%arg1, en%arg2)
      end if
   end if
```

```
case (EN_EVAL_FUN_UNARY)
   call eval_node_evaluate (en%arg1)
   en%value_is_known = prt_list_is_nonempty (en%arg1%pval)
   if (en%value_is_known) then
      en%arg0%index => en%index
      en%index = 1
      en%arg0%prt1 => en%prt1
      en%prt1 = prt_list_get_prt (en%arg1%pval, 1)
      call eval_node_evaluate (en%arg0)
      en%rval = en%arg0%rval
   end if
case (EN_EVAL_FUN_BINARY)
   call eval_node_evaluate (en%arg1)
   call eval_node_evaluate (en%arg2)
   en%value_is_known = &
         prt_list_is_nonempty (en%arg1%pval) .and. &
         prt_list_is_nonempty (en%arg2%pval)
   if (en%value_is_known) then
      en%arg0%index => en%index
      en%arg0%prt1 => en%prt1
      en%arg0%prt2 => en%prt2
      en%index = 1
      en%prt1 = prt_list_get_prt (en%arg1%pval, 1)
      en%prt2 = prt_list_get_prt (en%arg2%pval, 1)
      call eval_node_evaluate (en%arg0)
      en%rval = en%arg0%rval
   end if
case (EN_LOG_FUN_UNARY)
   call eval_node_evaluate (en%arg1)
   en%value_is_known = .true.
   if (en%value_is_known) then
      en%arg0%index => en%index
      en%arg0%prt1 => en%prt1
      en%lval = en% op1_cut (en%arg1, en%arg0)
   end if
case (EN_LOG_FUN_BINARY)
   call eval_node_evaluate (en%arg1)
   call eval_node_evaluate (en%arg2)
   en%value_is_known = .true.
   if (en%value_is_known) then
      en%arg0%index => en%index
      en%arg0%prt1 => en%prt1
      en%arg0%prt2 => en%prt2
      en%lval = en% op2_cut (en%arg1, en%arg2, en%arg0)
   end if
case (EN_INT_FUN_UNARY)
   call eval_node_evaluate (en%arg1)
   en%value_is_known = en%arg1%value_is_known
   if (en%value_is_known) then
      if (associated (en%arg0)) then
         en%arg0%index => en%index
         en%arg0%prt1 => en%prt1
         call en% op1_num (en%ival, en%arg1, en%arg0)
      else
```

```
                  call en% op1_num (en%ival, en%arg1)
               end if
            end if
      case (EN_INT_FUN_BINARY)
         call eval_node_evaluate (en%arg1)
         call eval_node_evaluate (en%arg2)
         en%value_is_known = &
               en%arg1%value_is_known .and. &
               en%arg2%value_is_known
         if (en%value_is_known) then
            if (associated (en%arg0)) then
               en%arg0%index => en%index
               en%arg0%prt1 => en%prt1
               en%arg0%prt2 => en%prt2
               call en% op2_num (en%ival, en%arg1, en%arg2, en%arg0)
            else
               call en% op2_num (en%ival, en%arg1, en%arg2)
            end if
         end if
      case (EN_FORMAT_STR)
         if (associated (en%arg0)) then
            call eval_node_evaluate (en%arg0)
            en%value_is_known = en%arg0%value_is_known
         else
            en%value_is_known = .true.
         end if
         if (associated (en%arg1)) then
            call eval_node_evaluate (en%arg1)
            en%value_is_known = &
                  en%value_is_known .and. en%arg1%value_is_known
            if (en%value_is_known) then
               call evaluate_sprintf (en%sval, en%ival, en%arg0, en%arg1)
            end if
         else
            if (en%value_is_known) then
               call evaluate_sprintf (en%sval, en%ival, en%arg0)
            end if
         end if
      end select
      if (debug) then
         print *, "evaluated"
         call eval_node_write (en)
      end if
   end subroutine eval_node_evaluate
```

5.5.7 Evaluation syntax

We have two different flavors of the syntax: with and without particles.

⟨*Expressions: variables*⟩+≡

```
    type(syntax_t), target, save :: syntax_expr
    type(syntax_t), target, save :: syntax_pexpr
```

These are for testing only and may be removed:

⟨*Expressions: public*⟩≡
```
    public :: syntax_expr_init
    public :: syntax_pexpr_init
```

⟨*Expressions: procedures*⟩+≡
```
    subroutine syntax_expr_init ()
      type(ifile_t) :: ifile
      call define_expr_syntax (ifile, particles=.false., analysis=.false.)
      call syntax_init (syntax_expr, ifile)
      call ifile_final (ifile)
    end subroutine syntax_expr_init

    subroutine syntax_pexpr_init ()
      type(ifile_t) :: ifile
      call define_expr_syntax (ifile, particles=.true., analysis=.false.)
      call syntax_init (syntax_pexpr, ifile)
      call ifile_final (ifile)
    end subroutine syntax_pexpr_init
```

⟨*Expressions: public*⟩+≡
```
    public :: syntax_expr_final
    public :: syntax_pexpr_final
```

⟨*Expressions: procedures*⟩+≡
```
    subroutine syntax_expr_final ()
      call syntax_final (syntax_expr)
    end subroutine syntax_expr_final

    subroutine syntax_pexpr_final ()
      call syntax_final (syntax_pexpr)
    end subroutine syntax_pexpr_final
```

⟨*Expressions: public*⟩+≡
```
    public :: syntax_pexpr_write
```

⟨*Expressions: procedures*⟩+≡
```
    subroutine syntax_pexpr_write (unit)
      integer, intent(in), optional :: unit
      call syntax_write (syntax_pexpr, unit)
    end subroutine syntax_pexpr_write
```

⟨*Expressions: public*⟩+≡
```
    public :: define_expr_syntax
```

Numeric expressions.

⟨*Expressions: procedures*⟩+≡
```
    subroutine define_expr_syntax (ifile, particles, analysis)
      type(ifile_t), intent(inout) :: ifile
      logical, intent(in) :: particles, analysis
      type(string_t) :: numeric_pexpr
      type(string_t) :: var_plist, var_alias
      if (particles) then
         numeric_pexpr = " | numeric_pexpr"
         var_plist = " | var_plist"
```

```
      var_alias = " | var_alias"
else
      numeric_pexpr = ""
      var_plist = ""
      var_alias = ""
end if
call ifile_append (ifile, "SEQ expr = subexpr addition*")
call ifile_append (ifile, "ALT subexpr = addition | term")
call ifile_append (ifile, "SEQ addition = plus_or_minus term")
call ifile_append (ifile, "SEQ term = factor multiplication*")
call ifile_append (ifile, "SEQ multiplication = times_or_over factor")
call ifile_append (ifile, "SEQ factor = value exponentiation?")
call ifile_append (ifile, "SEQ exponentiation = to_the value")
call ifile_append (ifile, "ALT plus_or_minus = '+' | '-'")
call ifile_append (ifile, "ALT times_or_over = '*' | '/'")
call ifile_append (ifile, "ALT to_the = '^' | '**'")
call ifile_append (ifile, "KEY '+'")
call ifile_append (ifile, "KEY '-'")
call ifile_append (ifile, "KEY '*'")
call ifile_append (ifile, "KEY '/'")
call ifile_append (ifile, "KEY '^'")
call ifile_append (ifile, "KEY '**'")
call ifile_append (ifile, "ALT value = signed_value | unsigned_value")
call ifile_append (ifile, "SEQ signed_value = '-' unsigned_value")
call ifile_append (ifile, "ALT unsigned_value = " // &
      "numeric_value | constant | variable | result | " // &
      "grouped_expr | block_expr | conditional_expr | " // &
      "unary_function | binary_function" // &
      numeric_pexpr)
call ifile_append (ifile, "ALT numeric_value = integer_value | " &
      // "real_value | complex_value")
call ifile_append (ifile, "SEQ integer_value = integer_literal unit_expr?")
call ifile_append (ifile, "SEQ real_value = real_literal unit_expr?")
call ifile_append (ifile, "SEQ complex_value = complex_literal unit_expr?")
call ifile_append (ifile, "INT integer_literal")
call ifile_append (ifile, "REA real_literal")
call ifile_append (ifile, "COM complex_literal")
call ifile_append (ifile, "SEQ unit_expr = unit unit_power?")
call ifile_append (ifile, "ALT unit = " // &
      "TeV | GeV | MeV | keV | eV | meV | " // &
      "nbarn | pbarn | fbarn | abarn | " // &
      "rad | mrad | degree | '%'")
call ifile_append (ifile, "KEY TeV")
call ifile_append (ifile, "KEY GeV")
call ifile_append (ifile, "KEY MeV")
call ifile_append (ifile, "KEY keV")
call ifile_append (ifile, "KEY eV")
call ifile_append (ifile, "KEY meV")
call ifile_append (ifile, "KEY nbarn")
call ifile_append (ifile, "KEY pbarn")
call ifile_append (ifile, "KEY fbarn")
call ifile_append (ifile, "KEY abarn")
call ifile_append (ifile, "KEY rad")
call ifile_append (ifile, "KEY mrad")
```

```
call ifile_append (ifile, "KEY degree")
call ifile_append (ifile, "KEY '%'")
call ifile_append (ifile, "SEQ unit_power = '^' frac_expr")
call ifile_append (ifile, "ALT frac_expr = frac | grouped_frac")
call ifile_append (ifile, "GRO grouped_frac = ( frac_expr )")
call ifile_append (ifile, "SEQ frac = signed_int div?")
call ifile_append (ifile, "ALT signed_int = " &
     // "neg_int | pos_int | integer_literal")
call ifile_append (ifile, "SEQ neg_int = '-' integer_literal")
call ifile_append (ifile, "SEQ pos_int = '+' integer_literal")
call ifile_append (ifile, "SEQ div = '/' integer_literal")
call ifile_append (ifile, "ALT constant = pi | I")
call ifile_append (ifile, "KEY pi")
call ifile_append (ifile, "KEY I")
call ifile_append (ifile, "IDE variable")
call ifile_append (ifile, "SEQ result = result_key result_arg")
call ifile_append (ifile, "ALT result_key = " // &
     "n_calls | integral | error | accuracy | efficiency | chi2")
call ifile_append (ifile, "KEY n_calls")
call ifile_append (ifile, "KEY integral")
call ifile_append (ifile, "KEY error")
call ifile_append (ifile, "KEY accuracy")
call ifile_append (ifile, "KEY efficiency")
call ifile_append (ifile, "KEY chi2")
call ifile_append (ifile, "GRO result_arg = ( process_id )")
call ifile_append (ifile, "IDE process_id")
call ifile_append (ifile, "SEQ unary_function = fun_unary function_arg1")
call ifile_append (ifile, "SEQ binary_function = fun_binary function_arg2")
call ifile_append (ifile, "ALT fun_unary = " // &
     "complex | real | int | nint | floor | ceiling | abs | sgn | " // &
     "sqrt | exp | log | log10 | " // &
     "sin | cos | tan | asin | acos | atan | " // &
     "sinh | cosh | tanh")
call ifile_append (ifile, "KEY complex")
call ifile_append (ifile, "KEY real")
call ifile_append (ifile, "KEY int")
call ifile_append (ifile, "KEY nint")
call ifile_append (ifile, "KEY floor")
call ifile_append (ifile, "KEY ceiling")
call ifile_append (ifile, "KEY abs")
call ifile_append (ifile, "KEY sgn")
call ifile_append (ifile, "KEY sqrt")
call ifile_append (ifile, "KEY exp")
call ifile_append (ifile, "KEY log")
call ifile_append (ifile, "KEY log10")
call ifile_append (ifile, "KEY sin")
call ifile_append (ifile, "KEY cos")
call ifile_append (ifile, "KEY tan")
call ifile_append (ifile, "KEY asin")
call ifile_append (ifile, "KEY acos")
call ifile_append (ifile, "KEY atan")
call ifile_append (ifile, "KEY sinh")
call ifile_append (ifile, "KEY cosh")
call ifile_append (ifile, "KEY tanh")
```

```
        call ifile_append (ifile, "ALT fun_binary = max | min | mod | modulo")
        call ifile_append (ifile, "KEY max")
        call ifile_append (ifile, "KEY min")
        call ifile_append (ifile, "KEY mod")
        call ifile_append (ifile, "KEY modulo")
        call ifile_append (ifile, "ARG function_arg1 = ( expr )")
        call ifile_append (ifile, "ARG function_arg2 = ( expr, expr )")
        call ifile_append (ifile, "GRO grouped_expr = ( expr )")
        call ifile_append (ifile, "SEQ block_expr = let var_spec in expr")
        call ifile_append (ifile, "KEY let")
        call ifile_append (ifile, "ALT var_spec = " // &
             "var_num | var_int | var_real | var_complex | " // &
             "var_logical" // var_plist // var_alias // " | var_string")
        call ifile_append (ifile, "SEQ var_num = var_name '=' expr")
        call ifile_append (ifile, "SEQ var_int = int var_name '=' expr")
        call ifile_append (ifile, "SEQ var_real = real var_name '=' expr")
        call ifile_append (ifile, "SEQ var_complex = complex var_name '=' complex_expr")
        call ifile_append (ifile, "ALT complex_expr = " // &
             "cexpr_real | cexpr_complex")
        call ifile_append (ifile, "ARG cexpr_complex = ( expr, expr )")
        call ifile_append (ifile, "SEQ cexpr_real = expr")
        call ifile_append (ifile, "IDE var_name")
        call ifile_append (ifile, "KEY '='")
        call ifile_append (ifile, "KEY in")
        call ifile_append (ifile, "SEQ conditional_expr = " // &
             "if lexpr then expr else expr endif")
        call ifile_append (ifile, "KEY if")
        call ifile_append (ifile, "KEY then")
        call ifile_append (ifile, "KEY else")
        call ifile_append (ifile, "KEY endif")
        call define_lexpr_syntax (ifile, particles, analysis)
        call define_sexpr_syntax (ifile)
        if (particles) then
           call define_pexpr_syntax (ifile)
           call define_cexpr_syntax (ifile)
           call define_var_plist_syntax (ifile)
           call define_var_alias_syntax (ifile)
           call define_numeric_pexpr_syntax (ifile)
           call define_logical_pexpr_syntax (ifile)
        end if

    end subroutine define_expr_syntax
```

Logical expressions.
⟨*Expressions: procedures*⟩+≡
```
    subroutine define_lexpr_syntax (ifile, particles, analysis)
      type(ifile_t), intent(inout) :: ifile
      logical, intent(in) :: particles, analysis
      type(string_t) :: logical_pexpr, record_cmd
      if (particles) then
         logical_pexpr = " | logical_pexpr"
      else
         logical_pexpr = ""
      end if
```

```
if (analysis) then
   record_cmd = " | record_cmd"
else
   record_cmd = ""
end if
call ifile_append (ifile, "SEQ lexpr = lterm alternative*")
call ifile_append (ifile, "SEQ alternative = or lterm")
call ifile_append (ifile, "SEQ lterm = lvalue coincidence*")
call ifile_append (ifile, "SEQ coincidence = and lvalue")
call ifile_append (ifile, "KEY or")
call ifile_append (ifile, "KEY and")
call ifile_append (ifile, "ALT lvalue = " // &
     "true | false | lvariable | negation | " // &
     "grouped_lexpr | block_lexpr | conditional_lexpr | " // &
     "compared_expr | compared_sexpr" // &
     logical_pexpr //  record_cmd)
call ifile_append (ifile, "KEY true")
call ifile_append (ifile, "KEY false")
call ifile_append (ifile, "SEQ lvariable = '?' alt_lvariable")
call ifile_append (ifile, "KEY '?'")
call ifile_append (ifile, "ALT alt_lvariable = variable | grouped_lexpr")
call ifile_append (ifile, "SEQ negation = not lvalue")
call ifile_append (ifile, "KEY not")
call ifile_append (ifile, "GRO grouped_lexpr = ( lexpr )")
call ifile_append (ifile, "SEQ block_lexpr = let var_spec in lexpr")
call ifile_append (ifile, "ALT var_logical = " // &
     "var_logical_new | var_logical_spec")
call ifile_append (ifile, "SEQ var_logical_new = logical var_logical_spec")
call ifile_append (ifile, "KEY logical")
call ifile_append (ifile, "SEQ var_logical_spec = '?' var_name = lexpr")
call ifile_append (ifile, "SEQ conditional_lexpr = " // &
     "if lexpr then lexpr else lexpr endif")
call ifile_append (ifile, "SEQ compared_expr = expr comparison+")
call ifile_append (ifile, "SEQ comparison = compare expr")
call ifile_append (ifile, "ALT compare = " // &
     "'<' | '>' | '<=' | '>=' | '==' | '<>' | '==~' | '<>~'")
call ifile_append (ifile, "KEY '<'")
call ifile_append (ifile, "KEY '>'")
call ifile_append (ifile, "KEY '<='")
call ifile_append (ifile, "KEY '>='")
call ifile_append (ifile, "KEY '=='")
call ifile_append (ifile, "KEY '<>'")
call ifile_append (ifile, "KEY '==~'")
call ifile_append (ifile, "KEY '<>~'")
call ifile_append (ifile, "SEQ compared_sexpr = sexpr str_comparison+")
call ifile_append (ifile, "SEQ str_comparison = str_compare sexpr")
call ifile_append (ifile, "ALT str_compare = '==' | '<>'")
if (analysis) then
   call ifile_append (ifile, "SEQ record_cmd = " // &
        "record_key analysis_tag record_arg?")
   call ifile_append (ifile, "ALT record_key = record | record_unweighted")
   call ifile_append (ifile, "KEY record")
   call ifile_append (ifile, "KEY record_unweighted")
   call ifile_append (ifile, "ALT analysis_tag = analysis_id | sexpr")
```

```
                 call ifile_append (ifile, "IDE analysis_id")
                 call ifile_append (ifile, "ARG record_arg = ( expr, expr? )")
             end if
         end subroutine define_lexpr_syntax
```

String expressions.

⟨*Expressions: procedures*⟩+≡
```
    subroutine define_sexpr_syntax (ifile)
      type(ifile_t), intent(inout) :: ifile
      call ifile_append (ifile, "SEQ sexpr = svalue str_concatenation*")
      call ifile_append (ifile, "SEQ str_concatenation = '//' svalue")
      call ifile_append (ifile, "KEY '//'")
      call ifile_append (ifile, "ALT svalue = " // &
           "grouped_sexpr | block_sexpr | conditional_sexpr | " // &
           "svariable | string_function | string_literal")
      call ifile_append (ifile, "GRO grouped_sexpr = ( sexpr )")
      call ifile_append (ifile, "SEQ block_sexpr = let var_spec in sexpr")
      call ifile_append (ifile, "SEQ conditional_sexpr = " // &
           "if lexpr then sexpr else sexpr endif")
      call ifile_append (ifile, "SEQ svariable = '$' alt_svariable")
      call ifile_append (ifile, "KEY '$'")
      call ifile_append (ifile, "ALT alt_svariable = variable | grouped_sexpr")
      call ifile_append (ifile, "ALT var_string = " // &
           "var_string_new | var_string_spec")
      call ifile_append (ifile, "SEQ var_string_new = string var_string_spec")
      call ifile_append (ifile, "KEY string")
      call ifile_append (ifile, "SEQ var_string_spec = '$' var_name = sexpr") ! $
      call ifile_append (ifile, "ALT string_function = sprintd_fun | sprintf_fun")
      call ifile_append (ifile, "SEQ sprintd_fun = sprint sprintf_args?")
      call ifile_append (ifile, "KEY sprint")
      call ifile_append (ifile, "SEQ sprintf_fun = sprintf_clause sprintf_args?")
      call ifile_append (ifile, "SEQ sprintf_clause = sprintf sexpr")
      call ifile_append (ifile, "KEY sprintf")
      call ifile_append (ifile, "ARG sprintf_args = ( sprintf_arg* )")
      call ifile_append (ifile, "ALT sprintf_arg = " &
           // "lvariable | svariable | expr")
      call ifile_append (ifile, "QUO string_literal = '""'...'""'")
    end subroutine define_sexpr_syntax
```

Expressions that evaluate to particle lists.

⟨*Expressions: procedures*⟩+≡
```
    subroutine define_pexpr_syntax (ifile)
      type(ifile_t), intent(inout) :: ifile
      call ifile_append (ifile, "SEQ pexpr = pvalue combination*")
      call ifile_append (ifile, "SEQ combination = '&' pvalue")
      call ifile_append (ifile, "KEY '&'")
      call ifile_append (ifile, "ALT pvalue = " // &
           "pexpr_src | pvariable | " // &
           "grouped_pexpr | block_pexpr | conditional_pexpr | " // &
           "prt_function")
      call ifile_append (ifile, "SEQ pexpr_src = prefix_cexpr")
      call ifile_append (ifile, "ALT prefix_cexpr = " // &
           "incoming_prt | outgoing_prt")
```

```
      call ifile_append (ifile, "SEQ incoming_prt = incoming cexpr")
      call ifile_append (ifile, "KEY incoming")
      call ifile_append (ifile, "SEQ outgoing_prt = cexpr")
      call ifile_append (ifile, "SEQ pvariable = '@' alt_pvariable")
      call ifile_append (ifile, "KEY '@'")
      call ifile_append (ifile, "ALT alt_pvariable = variable | grouped_pexpr")
      call ifile_append (ifile, "GRO grouped_pexpr = '[' pexpr ']'")
      call ifile_append (ifile, "SEQ block_pexpr = let var_spec in pexpr")
      call ifile_append (ifile, "SEQ conditional_pexpr = " // &
           "if lexpr then pexpr else pexpr endif")
      call ifile_append (ifile, "ALT prt_function = " // &
           "join_fun | combine_fun | collect_fun | select_fun | " // &
           "extract_fun | sort_fun")
      call ifile_append (ifile, "SEQ join_fun = join_clause pargs2")
      call ifile_append (ifile, "SEQ combine_fun = combine_clause pargs2")
      call ifile_append (ifile, "SEQ collect_fun = collect_clause pargs1")
      call ifile_append (ifile, "SEQ select_fun = select_clause pargs1")
      call ifile_append (ifile, "SEQ extract_fun = extract_clause pargs1")
      call ifile_append (ifile, "SEQ sort_fun = sort_clause pargs1")
      call ifile_append (ifile, "SEQ join_clause = join condition?")
      call ifile_append (ifile, "SEQ combine_clause = combine condition?")
      call ifile_append (ifile, "SEQ collect_clause = collect condition?")
      call ifile_append (ifile, "SEQ select_clause = select condition?")
      call ifile_append (ifile, "SEQ extract_clause = extract position?")
      call ifile_append (ifile, "SEQ sort_clause = sort criterion?")
      call ifile_append (ifile, "KEY join")
      call ifile_append (ifile, "KEY combine")
      call ifile_append (ifile, "KEY collect")
      call ifile_append (ifile, "KEY select")
      call ifile_append (ifile, "SEQ condition = if lexpr")
      call ifile_append (ifile, "KEY extract")
      call ifile_append (ifile, "SEQ position = index expr")
      call ifile_append (ifile, "KEY sort")
      call ifile_append (ifile, "SEQ criterion = by expr")
      call ifile_append (ifile, "KEY index")
      call ifile_append (ifile, "KEY by")
      call ifile_append (ifile, "ARG pargs2 = '[' pexpr, pexpr ']'")
      call ifile_append (ifile, "ARG pargs1 = '[' pexpr, pexpr? ']'")
    end subroutine define_pexpr_syntax
```

Expressions that evaluate to PDG-code arrays.
⟨*Expressions: procedures*⟩+≡

```
    subroutine define_cexpr_syntax (ifile)
      type(ifile_t), intent(inout) :: ifile
      call ifile_append (ifile, "SEQ cexpr = avalue concatenation*")
      call ifile_append (ifile, "SEQ concatenation = ':' avalue")
      call ifile_append (ifile, "KEY ':'")
      call ifile_append (ifile, "ALT avalue = " // &
           "grouped_cexpr | block_cexpr | conditional_cexpr | " // &
           "variable | pdg_code | prt_name")
      call ifile_append (ifile, "GRO grouped_cexpr = ( cexpr )")
      call ifile_append (ifile, "SEQ block_cexpr = let var_spec in cexpr")
      call ifile_append (ifile, "SEQ conditional_cexpr = " // &
           "if lexpr then cexpr else cexpr endif")
```

```
          call ifile_append (ifile, "SEQ pdg_code = pdg pdg_arg")
          call ifile_append (ifile, "KEY pdg")
          call ifile_append (ifile, "ARG pdg_arg = ( expr )")
          call ifile_append (ifile, "QUO prt_name = '""'...'""'")
      end subroutine define_cexpr_syntax
```

Extra variable types.

⟨*Expressions: procedures*⟩+≡
```
      subroutine define_var_plist_syntax (ifile)
        type(ifile_t), intent(inout) :: ifile
        call ifile_append (ifile, "ALT var_plist = var_plist_new | var_plist_spec")
        call ifile_append (ifile, "SEQ var_plist_new = subevt var_plist_spec")
        call ifile_append (ifile, "KEY subevt")
        call ifile_append (ifile, "SEQ var_plist_spec = '@' var_name '=' pexpr")
      end subroutine define_var_plist_syntax

      subroutine define_var_alias_syntax (ifile)
        type(ifile_t), intent(inout) :: ifile
        call ifile_append (ifile, "SEQ var_alias = alias var_name '=' cexpr")
        call ifile_append (ifile, "KEY alias")
      end subroutine define_var_alias_syntax
```

Particle-list expressions that evaluate to numeric values

⟨*Expressions: procedures*⟩+≡
```
      subroutine define_numeric_pexpr_syntax (ifile)
        type(ifile_t), intent(inout) :: ifile
        call ifile_append (ifile, "ALT numeric_pexpr = eval_fun | count_fun")
        call ifile_append (ifile, "SEQ eval_fun = eval expr pargs1")
        call ifile_append (ifile, "SEQ count_fun = count_clause pargs1")
        call ifile_append (ifile, "SEQ count_clause = count condition?")
        call ifile_append (ifile, "KEY eval")
        call ifile_append (ifile, "KEY count")
      end subroutine define_numeric_pexpr_syntax
```

Particle-list functions that evaluate to logical values.

⟨*Expressions: procedures*⟩+≡
```
      subroutine define_logical_pexpr_syntax (ifile)
        type(ifile_t), intent(inout) :: ifile
        call ifile_append (ifile, "ALT logical_pexpr = " // &
            "all_fun | any_fun | no_fun")
        call ifile_append (ifile, "SEQ all_fun = all lexpr pargs1")
        call ifile_append (ifile, "SEQ any_fun = any lexpr pargs1")
        call ifile_append (ifile, "SEQ no_fun = no lexpr pargs1")
        call ifile_append (ifile, "KEY all")
        call ifile_append (ifile, "KEY any")
        call ifile_append (ifile, "KEY no")
      end subroutine define_logical_pexpr_syntax
```

All characters that can occur in expressions (apart from alphanumeric).

⟨*Expressions: procedures*⟩+≡
```
      subroutine lexer_init_eval_tree (lexer, particles)
        type(lexer_t), intent(out) :: lexer
```

```
    logical, intent(in) :: particles
    type(keyword_list_t), pointer :: keyword_list
    if (particles) then
        keyword_list => syntax_get_keyword_list_ptr (syntax_pexpr)
    else
        keyword_list => syntax_get_keyword_list_ptr (syntax_expr)
    end if
    call lexer_init (lexer, &
        comment_chars = "#!", &
        quote_chars = '"', &
        quote_match = '"', &
        single_chars = "()[],:&%?$@", &
        special_class = (/ "+-*/^", "<>=~ " /) , &
        keyword_list = keyword_list)
end subroutine lexer_init_eval_tree
```

5.5.8   Set up appropriate parse trees

Parse an input stream as a specific flavor of expression. The appropriate expression syntax has to be available.

⟨*Expressions: public*⟩+≡
```
    public :: parse_tree_init_expr
    public :: parse_tree_init_lexpr
    public :: parse_tree_init_pexpr
    public :: parse_tree_init_cexpr
    public :: parse_tree_init_sexpr
```

⟨*Expressions: procedures*⟩+≡
```
    subroutine parse_tree_init_expr (parse_tree, stream, particles)
      type(parse_tree_t), intent(out) :: parse_tree
      type(stream_t), intent(inout), target :: stream
      logical, intent(in) :: particles
      type(lexer_t) :: lexer
      call lexer_init_eval_tree (lexer, particles)
      call lexer_assign_stream (lexer, stream)
      if (particles) then
         call parse_tree_init &
              (parse_tree, syntax_pexpr, lexer, var_str ("expr"))
      else
         call parse_tree_init &
              (parse_tree, syntax_expr, lexer, var_str ("expr"))
      end if
      call lexer_final (lexer)
    end subroutine parse_tree_init_expr

    subroutine parse_tree_init_lexpr (parse_tree, stream, particles)
      type(parse_tree_t), intent(out) :: parse_tree
      type(stream_t), intent(inout), target :: stream
      logical, intent(in) :: particles
      type(lexer_t) :: lexer
      call lexer_init_eval_tree (lexer, particles)
      call lexer_assign_stream (lexer, stream)
      if (particles) then
```

```
      call parse_tree_init &
           (parse_tree, syntax_pexpr, lexer, var_str ("lexpr"))
    else
       call parse_tree_init &
            (parse_tree, syntax_expr, lexer, var_str ("lexpr"))
    end if
    call lexer_final (lexer)
  end subroutine parse_tree_init_lexpr

  subroutine parse_tree_init_pexpr (parse_tree, stream)
    type(parse_tree_t), intent(out) :: parse_tree
    type(stream_t), intent(inout), target :: stream
    type(lexer_t) :: lexer
    call lexer_init_eval_tree (lexer, .true.)
    call lexer_assign_stream (lexer, stream)
    call parse_tree_init &
         (parse_tree, syntax_pexpr, lexer, var_str ("pexpr"))
    call lexer_final (lexer)
  end subroutine parse_tree_init_pexpr

  subroutine parse_tree_init_cexpr (parse_tree, stream)
    type(parse_tree_t), intent(out) :: parse_tree
    type(stream_t), intent(inout), target :: stream
    type(lexer_t) :: lexer
    call lexer_init_eval_tree (lexer, .true.)
    call lexer_assign_stream (lexer, stream)
    call parse_tree_init &
         (parse_tree, syntax_pexpr, lexer, var_str ("cexpr"))
    call lexer_final (lexer)
  end subroutine parse_tree_init_cexpr

  subroutine parse_tree_init_sexpr (parse_tree, stream, particles)
    type(parse_tree_t), intent(out) :: parse_tree
    type(stream_t), intent(inout), target :: stream
    logical, intent(in) :: particles
    type(lexer_t) :: lexer
    call lexer_init_eval_tree (lexer, particles)
    call lexer_assign_stream (lexer, stream)
    if (particles) then
       call parse_tree_init &
            (parse_tree, syntax_pexpr, lexer, var_str ("sexpr"))
    else
       call parse_tree_init &
            (parse_tree, syntax_expr, lexer, var_str ("sexpr"))
    end if
    call lexer_final (lexer)
  end subroutine parse_tree_init_sexpr
```

5.5.9   The evaluation tree

The evaluation tree contains the initial variable list and the root node.

⟨*Expressions: public*⟩+≡

```
      public :: eval_tree_t
```
⟨*Expressions: types*⟩+≡
```
    type :: eval_tree_t
       private
       type(var_list_t) :: var_list
       type(eval_node_t), pointer :: root => null ()
    end type eval_tree_t
```

Init from stream, using a temporary parse tree.

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_tree_init_stream &
         (eval_tree, stream, var_list, prt_list, result_type)
      type(eval_tree_t), intent(out), target :: eval_tree
      type(stream_t), intent(inout), target :: stream
      type(var_list_t), intent(in), target :: var_list
      type(prt_list_t), intent(in), target, optional :: prt_list
      integer, intent(in), optional :: result_type
      type(parse_tree_t) :: parse_tree
      type(parse_node_t), pointer :: nd_root
      integer :: type
      type = V_REAL;  if (present (result_type))  type = result_type
      select case (type)
      case (V_INT, V_REAL, V_CMPLX)
         call parse_tree_init_expr (parse_tree, stream, present (prt_list))
      case (V_LOG)
         call parse_tree_init_lexpr (parse_tree, stream, present (prt_list))
      case (V_PTL)
         call parse_tree_init_pexpr (parse_tree, stream)
      case (V_PDG)
         call parse_tree_init_cexpr (parse_tree, stream)
      case (V_STR)
         call parse_tree_init_sexpr (parse_tree, stream, present (prt_list))
      end select
      call parse_tree_write (parse_tree)
      nd_root => parse_tree_get_root_ptr (parse_tree)
      if (associated (nd_root)) then
         select case (type)
         case (V_INT, V_REAL, V_CMPLX)
            call eval_tree_init_expr (eval_tree, nd_root, var_list, prt_list)
         case (V_LOG)
            call eval_tree_init_lexpr (eval_tree, nd_root, var_list, prt_list)
         case (V_PTL)
            call eval_tree_init_pexpr (eval_tree, nd_root, var_list, prt_list)
         case (V_PDG)
            call eval_tree_init_cexpr (eval_tree, nd_root, var_list, prt_list)
         case (V_STR)
            call eval_tree_init_sexpr (eval_tree, nd_root, var_list, prt_list)
         end select
      end if
      call parse_tree_final (parse_tree)
    end subroutine eval_tree_init_stream
```

API: Init from a given parse-tree node. If we evaluate an expression that contains particle-list references, the original particle list has to be supplied. The initial variable list is optional.

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_init_expr
    public :: eval_tree_init_lexpr
    public :: eval_tree_init_pexpr
    public :: eval_tree_init_cexpr
    public :: eval_tree_init_sexpr
```

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_tree_init_expr &
        (eval_tree, parse_node, var_list, prt_list, event_weight)
      type(eval_tree_t), intent(out), target :: eval_tree
      type(parse_node_t), intent(in), target :: parse_node
      type(var_list_t), intent(in), target :: var_list
      type(prt_list_t), intent(in), optional, target :: prt_list
      real(default), intent(in), optional, target :: event_weight
      call eval_tree_set_var_list (eval_tree, var_list, prt_list, event_weight)
      call eval_node_compile_expr &
          (eval_tree%root, parse_node, eval_tree%var_list)
    end subroutine eval_tree_init_expr

    subroutine eval_tree_init_lexpr &
        (eval_tree, parse_node, var_list, prt_list, event_weight)
      type(eval_tree_t), intent(out), target :: eval_tree
      type(parse_node_t), intent(in), target :: parse_node
      type(var_list_t), intent(in), target :: var_list
      type(prt_list_t), intent(in), optional, target :: prt_list
      real(default), intent(in), optional, target :: event_weight
      call eval_tree_set_var_list (eval_tree, var_list, prt_list, event_weight)
      call eval_node_compile_lexpr &
          (eval_tree%root, parse_node, eval_tree%var_list)
    end subroutine eval_tree_init_lexpr

    subroutine eval_tree_init_pexpr &
        (eval_tree, parse_node, var_list, prt_list, event_weight)
      type(eval_tree_t), intent(out), target :: eval_tree
      type(parse_node_t), intent(in), target :: parse_node
      type(var_list_t), intent(in), target :: var_list
      type(prt_list_t), intent(in), optional, target :: prt_list
      real(default), intent(in), optional, target :: event_weight
      call eval_tree_set_var_list (eval_tree, var_list, prt_list, event_weight)
      call eval_node_compile_pexpr &
          (eval_tree%root, parse_node, eval_tree%var_list)
    end subroutine eval_tree_init_pexpr

    subroutine eval_tree_init_cexpr &
        (eval_tree, parse_node, var_list, prt_list, event_weight)
      type(eval_tree_t), intent(out), target :: eval_tree
      type(parse_node_t), intent(in), target :: parse_node
      type(var_list_t), intent(in), target :: var_list
      type(prt_list_t), intent(in), optional, target :: prt_list
      real(default), intent(in), optional, target :: event_weight
```

```
      call eval_tree_set_var_list (eval_tree, var_list, prt_list, event_weight)
      call eval_node_compile_cexpr &
           (eval_tree%root, parse_node, eval_tree%var_list)
    end subroutine eval_tree_init_cexpr

    subroutine eval_tree_init_sexpr &
         (eval_tree, parse_node, var_list, prt_list, event_weight)
      type(eval_tree_t), intent(out), target :: eval_tree
      type(parse_node_t), intent(in), target :: parse_node
      type(var_list_t), intent(in), target :: var_list
      type(prt_list_t), intent(in), optional, target :: prt_list
      real(default), intent(in), optional, target :: event_weight
      call eval_tree_set_var_list (eval_tree, var_list, prt_list, event_weight)
      call eval_node_compile_sexpr &
           (eval_tree%root, parse_node, eval_tree%var_list)
    end subroutine eval_tree_init_sexpr
```

This extra API function handles numerical constant expressions only. The only
nontrivial part is the optional unit.

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_init_numeric_value
```

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_tree_init_numeric_value (eval_tree, parse_node)
      type(eval_tree_t), intent(out), target :: eval_tree
      type(parse_node_t), intent(in), target :: parse_node
      call eval_node_compile_numeric_value (eval_tree%root, parse_node)
    end subroutine eval_tree_init_numeric_value
```

Initialize the variable list with the initial one; if a particle list is provided, add a
pointer to this as variable @evt. If the event weight is provided as a real-valued
target, add a pointer to it as well.

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_tree_set_var_list &
         (eval_tree, var_list, prt_list, event_weight)
      type(eval_tree_t), intent(inout), target :: eval_tree
      type(var_list_t), intent(in), target :: var_list
      type(prt_list_t), intent(in), optional, target :: prt_list
      real(default), intent(in), optional, target :: event_weight
      logical, save, target :: known = .true.
      call var_list_link (eval_tree%var_list, var_list)
      if (present (prt_list))  call var_list_append_prt_list_ptr &
           (eval_tree%var_list, var_str ("@evt"), prt_list, known, &
           intrinsic=.true.)
      if (present (event_weight))  call var_list_append_real_ptr &
           (eval_tree%var_list, var_str ("event_weight"), event_weight, known, &
            intrinsic=.true.)
    end subroutine eval_tree_set_var_list
```

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_final
```

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_tree_final (eval_tree)
      type(eval_tree_t), intent(inout) :: eval_tree
      call var_list_final (eval_tree%var_list)
      if (associated (eval_tree%root)) then
         call eval_node_final_rec (eval_tree%root)
         deallocate (eval_tree%root)
      end if
    end subroutine eval_tree_final
```

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_evaluate
```

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_tree_evaluate (eval_tree)
      type(eval_tree_t), intent(inout) :: eval_tree
      if (associated (eval_tree%root)) then
         call eval_node_evaluate (eval_tree%root)
      end if
    end subroutine eval_tree_evaluate
```

Check if the eval tree is allocated.

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_is_defined
```

⟨*Expressions: procedures*⟩+≡
```
    function eval_tree_is_defined (eval_tree) result (flag)
      logical :: flag
      type(eval_tree_t), intent(in) :: eval_tree
      flag = associated (eval_tree%root)
    end function eval_tree_is_defined
```

Check if the eval tree result is constant.

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_is_constant
```

⟨*Expressions: procedures*⟩+≡
```
    function eval_tree_is_constant (eval_tree) result (flag)
      logical :: flag
      type(eval_tree_t), intent(in) :: eval_tree
      if (associated (eval_tree%root)) then
         flag = eval_tree%root%type == EN_CONSTANT
      else
         flag = .false.
      end if
    end function eval_tree_is_constant
```

Insert a conversion node at the root, if necessary (only for real/int conversion)

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_convert_result
```

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_tree_convert_result (eval_tree, result_type)
      type(eval_tree_t), intent(inout) :: eval_tree
      integer, intent(in) :: result_type
      if (associated (eval_tree%root)) then
          call insert_conversion_node (eval_tree%root, result_type)
      end if
    end subroutine eval_tree_convert_result
```

Return the value of the top node, after evaluation. If the tree is empty, return
the type of V_NONE. When extracting the value, no check for existence is done.
For numeric values, the functions are safe against real/integer mismatch.

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_get_result_type
    public :: eval_tree_result_is_known
    public :: eval_tree_result_is_known_ptr
    public :: eval_tree_get_log
    public :: eval_tree_get_int
    public :: eval_tree_get_real
    public :: eval_tree_get_cmplx
    public :: eval_tree_get_pdg_array
    public :: eval_tree_get_prt_list
    public :: eval_tree_get_string
```

⟨*Expressions: procedures*⟩+≡
```
    function eval_tree_get_result_type (eval_tree) result (type)
      integer :: type
      type(eval_tree_t), intent(in) :: eval_tree
      if (associated (eval_tree%root)) then
          type = eval_tree%root%result_type
      else
          type = V_NONE
      end if
    end function eval_tree_get_result_type

    function eval_tree_result_is_known (eval_tree) result (flag)
      logical :: flag
      type(eval_tree_t), intent(in) :: eval_tree
      if (associated (eval_tree%root)) then
          select case (eval_tree%root%result_type)
          case (V_LOG, V_INT, V_REAL)
              flag = eval_tree%root%value_is_known
          case default
              flag = .true.
          end select
      else
          flag = .false.
      end if
    end function eval_tree_result_is_known

    function eval_tree_result_is_known_ptr (eval_tree) result (ptr)
      logical, pointer :: ptr
      type(eval_tree_t), intent(in) :: eval_tree
      logical, target, save :: known = .true.
```

```
     if (associated (eval_tree%root)) then
        select case (eval_tree%root%result_type)
        case (V_LOG, V_INT, V_REAL)
           ptr => eval_tree%root%value_is_known
        case default
           ptr => known
        end select
     else
        ptr => null ()
     end if
end function eval_tree_result_is_known_ptr

function eval_tree_get_log (eval_tree) result (lval)
  logical :: lval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root))  lval = eval_tree%root%lval
end function eval_tree_get_log

function eval_tree_get_int (eval_tree) result (ival)
  integer :: ival
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     select case (eval_tree%root%result_type)
     case (V_INT);   ival = eval_tree%root%ival
     case (V_REAL); ival = eval_tree%root%rval
     case (V_CMPLX); ival = eval_tree%root%cval
     end select
  end if
end function eval_tree_get_int

function eval_tree_get_real (eval_tree) result (rval)
  real(default) :: rval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     select case (eval_tree%root%result_type)
     case (V_REAL); rval = eval_tree%root%rval
     case (V_INT);   rval = eval_tree%root%ival
     case (V_CMPLX);  rval = eval_tree%root%cval
     end select
  end if
end function eval_tree_get_real

function eval_tree_get_cmplx (eval_tree) result (cval)
  complex(default) :: cval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     select case (eval_tree%root%result_type)
     case (V_CMPLX); cval = eval_tree%root%cval
     case (V_REAL); cval = eval_tree%root%rval
     case (V_INT);  cval = eval_tree%root%ival
     end select
  end if
end function eval_tree_get_cmplx
```

```
function eval_tree_get_pdg_array (eval_tree) result (aval)
  type(pdg_array_t) :: aval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     aval = eval_tree%root%aval
  end if
end function eval_tree_get_pdg_array

function eval_tree_get_prt_list (eval_tree) result (pval)
  type(prt_list_t) :: pval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     pval = eval_tree%root%pval
  end if
end function eval_tree_get_prt_list

function eval_tree_get_string (eval_tree) result (sval)
  type(string_t) :: sval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     sval = eval_tree%root%sval
  end if
end function eval_tree_get_string
```

Return a pointer to the value of the top node.

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_get_log_ptr
    public :: eval_tree_get_int_ptr
    public :: eval_tree_get_real_ptr
    public :: eval_tree_get_cmplx_ptr
    public :: eval_tree_get_prt_list_ptr
    public :: eval_tree_get_pdg_array_ptr
    public :: eval_tree_get_string_ptr
```

⟨*Expressions: procedures*⟩+≡
```
    function eval_tree_get_log_ptr (eval_tree) result (lval)
      logical, pointer :: lval
      type(eval_tree_t), intent(in) :: eval_tree
      if (associated (eval_tree%root)) then
         lval => eval_tree%root%lval
      else
         lval => null ()
      end if
    end function eval_tree_get_log_ptr

    function eval_tree_get_int_ptr (eval_tree) result (ival)
      integer, pointer :: ival
      type(eval_tree_t), intent(in) :: eval_tree
      if (associated (eval_tree%root)) then
         ival => eval_tree%root%ival
      else
         ival => null ()
      end if
    end function eval_tree_get_int_ptr
```

449

```
function eval_tree_get_real_ptr (eval_tree) result (rval)
  real(default), pointer :: rval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     rval => eval_tree%root%rval
  else
     rval => null ()
  end if
end function eval_tree_get_real_ptr

function eval_tree_get_cmplx_ptr (eval_tree) result (cval)
  complex(default), pointer :: cval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     cval => eval_tree%root%cval
  else
     cval => null ()
  end if
end function eval_tree_get_cmplx_ptr

function eval_tree_get_prt_list_ptr (eval_tree) result (pval)
  type(prt_list_t), pointer :: pval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     pval => eval_tree%root%pval
  else
     pval => null ()
  end if
end function eval_tree_get_prt_list_ptr

function eval_tree_get_pdg_array_ptr (eval_tree) result (aval)
  type(pdg_array_t), pointer :: aval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     aval => eval_tree%root%aval
  else
     aval => null ()
  end if
end function eval_tree_get_pdg_array_ptr

function eval_tree_get_string_ptr (eval_tree) result (sval)
  type(string_t), pointer :: sval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     sval => eval_tree%root%sval
  else
     sval => null ()
  end if
end function eval_tree_get_string_ptr
```

⟨*Expressions: public*⟩+≡
```
public :: eval_tree_write
```

⟨*Expressions: procedures*⟩+≡

```
    subroutine eval_tree_write (eval_tree, unit, write_var_list)
      type(eval_tree_t), intent(in) :: eval_tree
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: write_var_list
      integer :: u
      logical :: vl
      u = output_unit (unit);  if (u < 0)  return
      vl = .false.;  if (present (write_var_list))  vl = write_var_list
      write (u, "(1x,A)") "Evaluation tree:"
      if (associated (eval_tree%root)) then
         call eval_node_write_rec (eval_tree%root, unit)
      else
         write (u, "(3x,A)") "[empty]"
      end if
      if (vl)  call var_list_write (eval_tree%var_list, unit)
    end subroutine eval_tree_write
```

5.5.10 Test

⟨*Expressions: public*⟩+≡

```
    public :: expressions_test
```

⟨*Expressions: procedures*⟩+≡

```
    subroutine expressions_test ()
      call expressions_test1 ()
  !     call syntax_expr_init ()
  !     call syntax_write (syntax_expr)
  !    call expressions_test1
  !     call syntax_expr_final ()
  !     print *
  !     call expressions_test2
  !     print *
  !     call syntax_pexpr_init ()
  !     call syntax_write (syntax_pexpr)
  !     call expressions_test3
  !     call syntax_pexpr_final ()
    end subroutine expressions_test

    subroutine expressions_test1 ()
      type(var_list_t) :: var_list
      type(eval_node_t) :: node
      type(prt_t), target :: prt
      type(var_entry_t), pointer :: var
      call var_list_set_observables_unary (var_list, prt)
      call var_list_write (var_list)
      var => var_list_get_var_ptr (var_list, var_str ("PDG"))
      call eval_node_init_obs (node, var)
      call eval_node_write (node)
    end subroutine expressions_test1

  !   subroutine expressions_test1 ()
  !     type(ifile_t) :: ifile
```

```
!     type(stream_t) :: stream
!     type(eval_tree_t) :: eval_tree
!     type(string_t) :: expr_text
!     type(var_list_t), target :: var_list
!     call var_list_append_real (var_list, var_str ("x"), -5._default)
!     call var_list_append_int  (var_list, var_str ("foo"), -27)
!     call var_list_append_real (var_list, var_str ("mb"), 4._default)
!     expr_text = &
!          "let real twopi = 2 * pi in" // &
!          " twopi * sqrt (25.d0 - mb^2)" // &
!          " / (let int mb_or_0 = max (mb, 0) in" // &
!          "       1 + (if -1 TeV <= x < mb_or_0 then abs(x) else x))"
!     call ifile_append (ifile, expr_text)
!     call stream_init (stream, ifile)
!     call var_list_write (var_list)
!     call eval_tree_init_stream (eval_tree, stream, var_list=var_list)
!     call eval_tree_evaluate (eval_tree)
!     call eval_tree_write (eval_tree)
!     print "(A)", "Input string:"
!     print *, char (expr_text)
!     call stream_final (stream)
!     call ifile_final (ifile)
!     call eval_tree_final (eval_tree)
!   end subroutine expressions_test1
!
!   subroutine expressions_test2 ()
!     type(prt_list_t) :: prt_list
!     call prt_list_init (prt_list)
!     call prt_list_reset (prt_list, 1)
!     call prt_list_set_incoming (prt_list, 1, &
!          22, vector4_moving (1.e3_default, 1.e3_default, 1), &
!          0._default, (/ 2 /))
!     call prt_list_write (prt_list)
!     call prt_list_reset (prt_list, 4)
!     call prt_list_reset (prt_list, 3)
!     call prt_list_set_incoming (prt_list, 1, &
!          21, vector4_moving (1.e3_default, 1.e3_default, 3), &
!          0._default, (/ 1 /))
!     call prt_list_polarize (prt_list, 1, -1)
!     call prt_list_set_outgoing (prt_list, 2, &
!          1, vector4_moving (0._default, 1.e3_default, 3), &
!          -1.e6_default, (/ 7 /))
!     call prt_list_set_composite (prt_list, 3, &
!          vector4_moving (-1.e3_default, 0._default, 3), &
!          (/ 2, 7 /))
!     call prt_list_write (prt_list)
!   end subroutine expressions_test2
!
!   subroutine expressions_test3 ()
!     type(prt_list_t), target :: prt_list
!     type(string_t) :: expr_text
!     type(ifile_t) :: ifile
!     type(stream_t) :: stream
!     type(eval_tree_t) :: eval_tree
```

452

```
!     type(var_list_t), target :: var_list
!     type(pdg_array_t) :: aval
!     aval = 0
!     call var_list_append_pdg_array (var_list, var_str ("particle"), aval)
!     aval = (/ 11,-11 /)
!     call var_list_append_pdg_array (var_list, var_str ("lepton"), aval)
!     aval = 22
!     call var_list_append_pdg_array (var_list, var_str ("photon"), aval)
!     aval = 1
!     call var_list_append_pdg_array (var_list, var_str ("u"), aval)
!     call prt_list_init (prt_list)
!     call prt_list_reset (prt_list, 6)
!     call prt_list_set_incoming (prt_list, 1, &
!          1, vector4_moving (1._default, 1._default, 1), 0._default)
!     call prt_list_set_incoming (prt_list, 2, &
!          -1, vector4_moving (2._default, 2._default, 1), 0._default)
!     call prt_list_set_outgoing (prt_list, 3, &
!          22, vector4_moving (3._default, 3._default, 1), 0._default)
!     call prt_list_set_outgoing (prt_list, 4, &
!          22, vector4_moving (4._default, 4._default, 1), 0._default)
!     call prt_list_set_outgoing (prt_list, 5, &
!          11, vector4_moving (5._default, 5._default, 1), 0._default)
!     call prt_list_set_outgoing (prt_list, 6, &
!          -11, vector4_moving (6._default, 6._default, 1), 0._default)
!     print *
!     print *, "Expression:"
!     expr_text = &
!          "let alias quark = pdg(1):pdg(2):pdg(3) in" // &
!          "  any E > 3 GeV " // &
!          "    (sort by - Pt " // &
!          "        (select if Index < 6 " // &
!          "           (outgoing photon:pdg(-11):pdg(3):quark " // &
!          "            & incoming particle)))" // &
!          "  and" // &
!          "  eval Theta (extract index -1 (outgoing photon)) > 45 degree" // &
!          "  and" // &
!          "  count (incoming photon) * 3 > 0"
!     print *, char (expr_text)
!     print *
!     call ifile_append (ifile, expr_text)
!     call stream_init (stream, ifile)
!     call eval_tree_init_stream (eval_tree, stream, var_list, prt_list, V_LOG)
!     print *
!     call eval_tree_write (eval_tree)
!     call eval_tree_evaluate (eval_tree)
!     print *
!     call eval_tree_write (eval_tree)
!     call stream_final (stream)
!     call ifile_final (ifile)
!     call eval_tree_final (eval_tree)
!   end subroutine expressions_test3
```

Chapter 6

Physics Models

While in previous WHIZARD versions, the physics model was partially hard-coded and injected into the main code via include files, the current version stores the accessible models in module variables. This allows for maintaining different models concurrently. Accessing the model is possible via pointers to the module variables; this is used by flavor objects, for instance.

6.1   Model module

⟨`models.f90`⟩≡
  ⟨*File header*⟩

  module models

    use iso_c_binding !NODEP!
  ⟨*Use kinds*⟩
    use kinds, only: i8, i32 !NODEP!
    use kinds, only: c_default_float !NODEP!
  ⟨*Use strings*⟩
    use limits, only: VERTEX_TABLE_SCALE_FACTOR !NODEP!
  ⟨*Use file utils*⟩
    use md5
    use os_interface
    use hashes, only: hash
    use diagnostics !NODEP!
    use ifiles
    use syntax_rules
    use lexers
    use parser
    use pdg_arrays
    use variables
    use expressions

  ⟨*Standard module head*⟩

  ⟨*Models: public*⟩

  ⟨*Models: parameters*⟩

⟨*Models: types*⟩

⟨*Models: interfaces*⟩

⟨*Models: variables*⟩

```
contains
```

⟨*Models: procedures*⟩

```
end module models
```

6.1.1   Physics Parameters

A parameter has a name, a value. Derived parameters also have a definition in terms of other parameters, which is stored as an `eval_tree`. External parameters are set by an external program.

⟨*Models: parameters*⟩≡
```
    integer, parameter :: PAR_NONE = 0
    integer, parameter :: PAR_INDEPENDENT = 1, PAR_DERIVED = 2
    integer, parameter :: PAR_EXTERNAL = 3
```

⟨*Models: public*⟩≡
```
    public :: parameter_t
```

⟨*Models: types*⟩≡
```
    type :: parameter_t
      private
      integer :: type  = PAR_NONE
      type(string_t) :: name
      real(default) :: value = 0
      type(eval_tree_t) :: eval_tree
    end type parameter_t
```

Initialization depends on parameter type. Independent parameters are initialized by a constant value or a constant numerical expression (which may contain a unit). Derived parameters are initialized by an arbitrary numerical expression, which makes use of the current variable list. The expression is evaluated by the function `parameter_reset`.

⟨*Models: procedures*⟩≡
```
    subroutine parameter_init_independent_value (par, name, value)
      type(parameter_t), intent(out) :: par
      type(string_t), intent(in) :: name
      real(default), intent(in) :: value
      par%type = PAR_INDEPENDENT
      par%name = name
      par%value = value
    end subroutine parameter_init_independent_value

    subroutine parameter_init_independent (par, name, pn)
      type(parameter_t), intent(out) :: par
```

```
      type(string_t), intent(in) :: name
      type(parse_node_t), intent(in), target :: pn
      par%type = PAR_INDEPENDENT
      par%name = name
      call eval_tree_init_numeric_value (par%eval_tree, pn)
      par%value = eval_tree_get_real (par%eval_tree)
    end subroutine parameter_init_independent

    subroutine parameter_init_derived (par, name, pn, var_list)
      type(parameter_t), intent(out) :: par
      type(string_t), intent(in) :: name
      type(parse_node_t), intent(in), target :: pn
      type(var_list_t), intent(in), target :: var_list
      par%type = PAR_DERIVED
      par%name = name
      call eval_tree_init_expr (par%eval_tree, pn, var_list=var_list)
      call parameter_reset_derived (par)
    end subroutine parameter_init_derived

    subroutine parameter_init_external (par, name)
      type(parameter_t), intent(out) :: par
      type(string_t), intent(in) :: name
      par%type = PAR_EXTERNAL
      par%name = name
    end subroutine parameter_init_external
```

The finalizer is needed for the evaluation tree in the definition.

⟨*Models: procedures*⟩+≡
```
    subroutine parameter_final (par)
      type(parameter_t), intent(inout) :: par
      call eval_tree_final (par%eval_tree)
    end subroutine parameter_final
```

All derived parameters should be recalculated if some independent parameters have changed:

⟨*Models: procedures*⟩+≡
```
    subroutine parameter_reset_derived (par)
      type(parameter_t), intent(inout) :: par
      select case (par%type)
      case (PAR_DERIVED)
         call eval_tree_evaluate (par%eval_tree)
         par%value = eval_tree_get_real (par%eval_tree)
      end select
    end subroutine parameter_reset_derived
```

Direct access to the parameter value:

⟨*Models: procedures*⟩+≡
```
    function parameter_get_value_ptr (par) result (val)
      real(default), pointer :: val
      type(parameter_t), intent(in), target :: par
      val => par%value
    end function parameter_get_value_ptr
```

456

Output. [We should have a formula format for the eval tree, suitable for input and output!]

⟨*Models: procedures*⟩+≡

```
subroutine parameter_write (par, unit, write_defs)
  type(parameter_t), intent(in) :: par
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: write_defs
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(3x,A)", advance="no")  "parameter"
  write (u, "(1x,A,1x,A)", advance="no")  char (par%name), "="
  write (u, "(G17.10)", advance="no") par%value
  select case (par%type)
  case (PAR_DERIVED)
     write (u, *) "  ! derived"
     if (present (write_defs)) then
        if (write_defs) then
           call eval_tree_write (par%eval_tree, unit)
        end if
     end if
  case (PAR_EXTERNAL)
     write (u, *) "  ! external"
  end select
end subroutine parameter_write
```

6.1.2 Particle codes

Let us define a few particle codes independent of the model.

Gauge bosons:

⟨*Models: types*⟩+≡

```
integer, parameter, public :: GLUON = 21
integer, parameter, public :: PHOTON = 22
integer, parameter, public :: Z_BOSON = 23
integer, parameter, public :: W_BOSON = 24
```

Hadrons:

⟨*Models: types*⟩+≡

```
integer, parameter, public :: PROTON = 2212
integer, parameter, public :: PION = 111
integer, parameter, public :: PIPLUS = 211
integer, parameter, public :: PIMINUS = - PIPLUS
```

Hadron remnants (internal)

⟨*Models: types*⟩+≡

```
integer, parameter, public :: HADRON_REMNANT = 90
integer, parameter, public :: HADRON_REMNANT_SINGLET = 91
integer, parameter, public :: HADRON_REMNANT_TRIPLET = 92
integer, parameter, public :: HADRON_REMNANT_OCTET = 93
```

Generic particles for internal use in event analysis:

⟨*Models: types*⟩+≡
```
    integer, parameter, public :: PRT_ANY = 81
    integer, parameter, public :: PRT_VISIBLE = 82
    integer, parameter, public :: PRT_CHARGED = 83
    integer, parameter, public :: PRT_COLORED = 84
```

Further particle codes for internal use:

⟨*Models: types*⟩+≡
```
    integer, parameter, public :: INVALID = 97
    integer, parameter, public :: KEYSTONE = 98
    integer, parameter, public :: COMPOSITE = 99
```

6.1.3 Spin codes

Somewhat redundant, but for better readability we define named constants for spin types. If the mass is nonzero, this is equal to the number of degrees of freedom.

⟨*Models: types*⟩+≡
```
    integer, parameter, public :: UNKNOWN=0
    integer, parameter, public :: SCALAR=1, SPINOR=2, VECTOR=3, &
         VECTORSPINOR=4, TENSOR=5
```

Isospin types and charge types are counted in an analogous way, where charge type 1 is charge 0, 2 is charge 1/3, and so on. Zero always means unknown. Note that charge and isospin types have an explicit sign.

Color types are defined as the dimension of the representation.

6.1.4 Particle data

The particle-data type holds all information that pertains to a particular particle within a particular model. Information such as spin type, particle code etc. is stored within the object itself, while mass and width are associated to parameters, otherwise assumed zero.

⟨*Models: public*⟩+≡
```
    public :: particle_data_t
```

⟨*Models: types*⟩+≡
```
    type :: particle_data_t
       private
       type(string_t) :: longname
       integer :: pdg = UNDEFINED
       logical :: is_visible = .true.
       logical :: is_parton = .false.
       logical :: is_gauge = .false.
       logical :: is_left_handed = .false.
       logical :: is_right_handed = .false.
       logical :: has_antiparticle = .false.
       logical :: p_is_stable = .true.
       logical :: p_decays_isotropically = .false.
       logical :: p_decays_diagonal = .false.
```

```
      logical :: a_is_stable = .true.
      logical :: a_decays_isotropically = .false.
      logical :: a_decays_diagonal = .false.
      type(string_t), dimension(:), allocatable :: name, anti
      type(string_t) :: tex_name, tex_anti
      integer :: spin_type = UNDEFINED
      integer :: isospin_type = 1
      integer :: charge_type = 1
      integer :: color_type = 1
      real(default), pointer :: mass_val => null ()
      type(parameter_t), pointer :: mass_src => null ()
      real(default), pointer :: width_val => null ()
      type(parameter_t), pointer :: width_src => null ()
      integer :: multiplicity = 1
   end type particle_data_t
```

Initialize particle data with PDG long name and PDG code. TEX names should
be initialized to avoid issues with accessing unallocated string contents.

⟨*Models: public*⟩+≡
```
    public :: particle_data_init
```

⟨*Models: procedures*⟩+≡
```
    subroutine particle_data_init (prt, longname, pdg)
      type(particle_data_t), intent(out) :: prt
      type(string_t), intent(in) :: longname
      integer, intent(in) :: pdg
      prt%longname = longname
      prt%pdg = pdg
      prt%tex_name = ""
      prt%tex_anti = ""
    end subroutine particle_data_init
```

Copy quantum numbers from another particle

⟨*Models: procedures*⟩+≡
```
    subroutine particle_data_copy (prt, prt_src)
      type(particle_data_t), intent(inout) :: prt
      type(particle_data_t), intent(in) :: prt_src
      prt%is_visible = prt_src%is_visible
      prt%is_parton = prt_src%is_parton
      prt%is_gauge = prt_src%is_gauge
      prt%is_left_handed = prt_src%is_left_handed
      prt%is_right_handed = prt_src%is_right_handed
      prt%spin_type = prt_src%spin_type
      prt%isospin_type = prt_src%isospin_type
      prt%charge_type = prt_src%charge_type
      prt%color_type = prt_src%color_type
      call particle_data_set_multiplicity (prt)
    end subroutine particle_data_copy
```

Set particle quantum numbers.

⟨*Models: public*⟩+≡
```
    public :: particle_data_set
```

⟨*Models: procedures*⟩+≡

```fortran
subroutine particle_data_set (prt, &
     is_visible, is_parton, is_gauge, is_left_handed, is_right_handed, &
     p_is_stable, p_decays_isotropically, p_decays_diagonal, &
     a_is_stable, a_decays_isotropically, a_decays_diagonal, &
     name, anti, tex_name, tex_anti, &
     spin_type, isospin_type, charge_type, color_type, &
     mass_src, width_src)
  type(particle_data_t), intent(inout) :: prt
  logical, intent(in), optional :: is_visible, is_parton, is_gauge
  logical, intent(in), optional :: is_left_handed, is_right_handed
  logical, intent(in), optional :: p_is_stable
  logical, intent(in), optional :: p_decays_isotropically, p_decays_diagonal
  logical, intent(in), optional :: a_is_stable
  logical, intent(in), optional :: a_decays_isotropically, a_decays_diagonal
  type(string_t), dimension(:), intent(in), optional :: name, anti
  type(string_t), intent(in), optional :: tex_name, tex_anti
  integer, intent(in), optional :: spin_type, isospin_type
  integer, intent(in), optional :: charge_type, color_type
  type(parameter_t), intent(in), target, optional :: mass_src, width_src
  if (present (is_visible))  prt%is_visible = is_visible
  if (present (is_parton))  prt%is_parton = is_parton
  if (present (is_gauge))  prt%is_gauge = is_gauge
  if (present (is_left_handed))  prt%is_left_handed = is_left_handed
  if (present (is_right_handed))  prt%is_right_handed = is_right_handed
  if (present (p_is_stable))  prt%p_is_stable = p_is_stable
  if (present (p_decays_isotropically)) &
       prt%p_decays_isotropically = p_decays_isotropically
  if (present (p_decays_diagonal)) &
       prt%p_decays_diagonal = p_decays_diagonal
  if (present (a_is_stable))  prt%a_is_stable = a_is_stable
  if (present (a_decays_isotropically)) &
       prt%a_decays_isotropically = a_decays_isotropically
  if (present (a_decays_diagonal)) &
       prt%a_decays_diagonal = a_decays_diagonal
  if (present (name)) then
     allocate (prt%name (size (name)))
     prt%name = name
  end if
  if (present (anti)) then
     allocate (prt%anti (size (anti)))
     prt%anti = anti
     prt%has_antiparticle = .true.
  end if
  if (present (tex_name))  prt%tex_name = tex_name
  if (present (tex_anti))  prt%tex_anti = tex_anti
  if (present (spin_type))  prt%spin_type = spin_type
  if (present (isospin_type))  prt%isospin_type = isospin_type
  if (present (charge_type))  prt%charge_type = charge_type
  if (present (color_type))  prt%color_type = color_type
  if (present (mass_src)) then
     prt%mass_src => mass_src
     prt%mass_val => parameter_get_value_ptr (mass_src)
  end if
```

```
      if (present (width_src)) then
         prt%width_src => width_src
         prt%width_val => parameter_get_value_ptr (width_src)
      end if
      if (present (spin_type) .or. present (mass_src)) then
         call particle_data_set_multiplicity (prt)
      end if
   end subroutine particle_data_set
```

Calculate the multiplicity given spin type and mass.

⟨*Models: procedures*⟩+≡
```
   subroutine particle_data_set_multiplicity (prt)
     type(particle_data_t), intent(inout) :: prt
     if (prt%spin_type /= SCALAR) then
        if (associated (prt%mass_src)) then
           prt%multiplicity = prt%spin_type
        else if (prt%is_left_handed .or. prt%is_right_handed) then
           prt%multiplicity = 1
        else
           prt%multiplicity = 2
        end if
     end if
   end subroutine particle_data_set_multiplicity
```

Set the mass/width value (not the pointer). The mass/width pointer must be
allocated.

⟨*Models: procedures*⟩+≡
```
   subroutine particle_data_set_mass (prt, mass)
     type(particle_data_t), intent(inout) :: prt
     real(default), intent(in) :: mass
     if (associated (prt%mass_val))  prt%mass_val = mass
   end subroutine particle_data_set_mass

   subroutine particle_data_set_width (prt, width)
     type(particle_data_t), intent(inout) :: prt
     real(default), intent(in) :: width
     if (associated (prt%width_val))  prt%width_val = width
   end subroutine particle_data_set_width
```

Loose ends

⟨*Models: public*⟩+≡
```
   public :: particle_data_freeze
```

⟨*Models: procedures*⟩+≡
```
   subroutine particle_data_freeze (prt)
     type(particle_data_t), intent(inout) :: prt
     if (.not. allocated (prt%name))  allocate (prt%name (0))
     if (.not. allocated (prt%anti))  allocate (prt%anti (0))
   end subroutine particle_data_freeze
```

Output

⟨*Models: procedures*⟩+≡

```
subroutine particle_data_write (prt, unit)
  type(particle_data_t), intent(in) :: prt
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(3x,A,1x,A)", advance="no") "particle", char (prt%longname)
  write (u, "(1x,I7)", advance="no") prt%pdg
  if (.not. prt%is_visible) write (u, "(3x,A)", advance="no") "invisible"
  if (prt%is_parton)  write (u, "(3x,A)", advance="no") "parton"
  if (prt%is_gauge)  write (u, "(3x,A)", advance="no") "gauge"
  if (prt%is_left_handed)  write (u, "(3x,A)", advance="no") "left"
  if (prt%is_right_handed)  write (u, "(3x,A)", advance="no") "right"
  write (u, *)
  write (u, "(5x,A)", advance="no") "name"
  if (allocated (prt%name)) then
     do i = 1, size (prt%name)
        write (u, "(1x,A)", advance="no")  '"' // char (prt%name(i)) // '"'
     end do
     write (u, *)
     if (prt%has_antiparticle) then
        write (u, "(5x,A)", advance="no") "anti"
        do i = 1, size (prt%anti)
           write (u, "(1x,A)", advance="no")  '"' // char (prt%anti(i)) // '"'
        end do
        write (u, *)
     end if
     if (prt%tex_name /= "") then
        write (u, "(5x,A)")  &
             "tex_name " // '"' // char (prt%tex_name) // '"'
     end if
     if (prt%has_antiparticle .and. prt%tex_anti /= "") then
        write (u, "(5x,A)")  &
             "tex_anti " // '"' // char (prt%tex_anti) // '"'
     end if
  else
     write (u, "(A)") "???"
  end if
  write (u, "(5x,A)", advance="no") "spin "
  select case (mod (prt%spin_type - 1, 2))
  case (0);  write (u, "(I1)", advance="no") (prt%spin_type-1) / 2
  case default;  write (u, "(I1,A)", advance="no") prt%spin_type-1, "/2"
  end select
!   write (u, "(3x,A,I1,A)") "! [multiplicity = ", prt%multiplicity, "]"
  if (abs (prt%isospin_type) /= 1) then
     write (u, "(5x,A)", advance="no") "isospin "
     select case (mod (abs (prt%isospin_type) - 1, 2))
     case (0);  write (u, "(I2)", advance="no") &
          sign (abs (prt%isospin_type) - 1, prt%isospin_type) / 2
     case default;  write (u, "(I2,A)", advance="no") &
          sign (abs (prt%isospin_type) - 1, prt%isospin_type), "/2"
     end select
  end if
  if (abs (prt%charge_type) /= 1) then
     write (u, "(5x,A)", advance="no") "charge "
```

```
            select case (mod (abs (prt%charge_type) - 1, 3))
            case (0);  write (u, "(I2)", advance="no") &
                  sign (abs (prt%charge_type) - 1, prt%charge_type) / 3
            case default;  write (u, "(I2,A)", advance="no") &
                  sign (abs (prt%charge_type) - 1, prt%charge_type), "/3"
            end select
          end if
          if (prt%color_type /= 1) then
            write (u, "(5x,A,I2)", advance="no") "color ", prt%color_type
          end if
          write (u, *)
          if (associated (prt%mass_src)) then
            write (u, "(5x,A)") "mass " // char (prt%mass_src%name)
            if (associated (prt%width_src)) then
               write (u, "(5x,A)") "width " // char (prt%width_src%name)
            end if
          end if
        end subroutine particle_data_write
```

Retrieve data:

⟨*Models: public*⟩+≡
```
    public :: particle_data_get_pdg
    public :: particle_data_get_pdg_anti
```

⟨*Models: procedures*⟩+≡
```
    elemental function particle_data_get_pdg (prt) result (pdg)
      integer :: pdg
      type(particle_data_t), intent(in) :: prt
      pdg = prt%pdg
    end function particle_data_get_pdg

    elemental function particle_data_get_pdg_anti (prt) result (pdg)
      integer :: pdg
      type(particle_data_t), intent(in) :: prt
      if (prt%has_antiparticle) then
         pdg = - prt%pdg
      else
         pdg = prt%pdg
      end if
    end function particle_data_get_pdg_anti
```

Predicates:

⟨*Models: public*⟩+≡
```
    public :: particle_data_is_visible
    public :: particle_data_is_parton
    public :: particle_data_is_gauge
    public :: particle_data_is_left_handed
    public :: particle_data_is_right_handed
    public :: particle_data_has_antiparticle
    public :: particle_data_is_stable
    public :: particle_data_decays_isotropically
    public :: particle_data_decays_diagonal
```

⟨*Models: procedures*⟩+≡

```
    elemental function particle_data_is_visible (prt) result (flag)
      logical :: flag
      type(particle_data_t), intent(in) :: prt
      flag = prt%is_visible
    end function particle_data_is_visible

    elemental function particle_data_is_parton (prt) result (flag)
      logical :: flag
      type(particle_data_t), intent(in) :: prt
      flag = prt%is_parton
    end function particle_data_is_parton

    elemental function particle_data_is_gauge (prt) result (flag)
      logical :: flag
      type(particle_data_t), intent(in) :: prt
      flag = prt%is_gauge
    end function particle_data_is_gauge

    elemental function particle_data_is_left_handed (prt) result (flag)
      logical :: flag
      type(particle_data_t), intent(in) :: prt
      flag = prt%is_left_handed
    end function particle_data_is_left_handed

    elemental function particle_data_is_right_handed (prt) result (flag)
      logical :: flag
      type(particle_data_t), intent(in) :: prt
      flag = prt%is_right_handed
    end function particle_data_is_right_handed

    elemental function particle_data_has_antiparticle (prt) result (flag)
      logical :: flag
      type(particle_data_t), intent(in) :: prt
      flag = prt%has_antiparticle
    end function particle_data_has_antiparticle

    elemental function particle_data_is_stable (prt, anti) result (flag)
      logical :: flag
      type(particle_data_t), intent(in) :: prt
      logical, intent(in), optional :: anti
      if (present (anti)) then
         if (anti) then
            flag = prt%a_is_stable
         else
            flag = prt%p_is_stable
         end if
      else
         flag = prt%p_is_stable
      end if
    end function particle_data_is_stable

    elemental function particle_data_decays_isotropically &
        (prt, anti) result (flag)
```

```
      logical :: flag
      type(particle_data_t), intent(in) :: prt
      logical, intent(in), optional :: anti
      if (present (anti)) then
         if (anti) then
            flag = prt%a_decays_isotropically
         else
            flag = prt%p_decays_isotropically
         end if
      else
         flag = prt%p_decays_isotropically
      end if
    end function particle_data_decays_isotropically

    elemental function particle_data_decays_diagonal &
         (prt, anti) result (flag)
      logical :: flag
      type(particle_data_t), intent(in) :: prt
      logical, intent(in), optional :: anti
      if (present (anti)) then
         if (anti) then
            flag = prt%a_decays_diagonal
         else
            flag = prt%p_decays_diagonal
         end if
      else
         flag = prt%p_decays_diagonal
      end if
    end function particle_data_decays_diagonal
```

Names. Return the first name in the list (or the first antiparticle name)

⟨*Models: public*⟩+≡
```
    public :: particle_data_get_name
```

⟨*Models: procedures*⟩+≡
```
    elemental function particle_data_get_name &
         (prt, is_antiparticle) result (name)
      type(string_t) :: name
      type(particle_data_t), intent(in) :: prt
      logical, intent(in) :: is_antiparticle
      if (is_antiparticle) then
         if (prt%has_antiparticle) then
            name = prt%anti(1)
         else
            name = prt%name(1)
         end if
      else
         name = prt%name(1)
      end if
    end function particle_data_get_name
```

Same for the TEX name.

⟨*Models: public*⟩+≡
```
    public :: particle_data_get_tex_name
```

⟨*Models: procedures*⟩+≡
```
    elemental function particle_data_get_tex_name &
        (prt, is_antiparticle) result (name)
      type(string_t) :: name
      type(particle_data_t), intent(in) :: prt
      logical, intent(in) :: is_antiparticle
      if (is_antiparticle) then
         if (prt%has_antiparticle) then
            name = prt%tex_anti
         else
            name = prt%tex_name
         end if
      else
         name = prt%tex_name
      end if
    end function particle_data_get_tex_name
```

Quantum numbers

⟨*Models: public*⟩+≡
```
    public :: particle_data_get_spin_type
    public :: particle_data_get_multiplicity
    public :: particle_data_get_isospin_type
    public :: particle_data_get_charge_type
    public :: particle_data_get_color_type
```

⟨*Models: procedures*⟩+≡
```
    elemental function particle_data_get_spin_type (prt) result (type)
      integer :: type
      type(particle_data_t), intent(in) :: prt
      type = prt%spin_type
    end function particle_data_get_spin_type

    elemental function particle_data_get_multiplicity (prt) result (type)
      integer :: type
      type(particle_data_t), intent(in) :: prt
      type = prt%multiplicity
    end function particle_data_get_multiplicity

    elemental function particle_data_get_isospin_type (prt) result (type)
      integer :: type
      type(particle_data_t), intent(in) :: prt
      type = prt%isospin_type
    end function particle_data_get_isospin_type

    elemental function particle_data_get_charge_type (prt) result (type)
      integer :: type
      type(particle_data_t), intent(in) :: prt
      type = prt%charge_type
    end function particle_data_get_charge_type

    elemental function particle_data_get_color_type (prt) result (type)
      integer :: type
      type(particle_data_t), intent(in) :: prt
      type = prt%color_type
```

```
    end function particle_data_get_color_type
```

In the MSSM, neutralinos can have a negative mass. This is relevant for computing matrix elements. However, within the WHIZARD main program we are interested only in kinematics, therefore we return the absolute value of the particle mass. If desired, we can extract the sign separately.

⟨*Models: public*⟩+≡
```
    public :: particle_data_get_charge
    public :: particle_data_get_mass
    public :: particle_data_get_mass_sign
    public :: particle_data_get_width
    public :: particle_data_get_isospin
```
⟨*Models: procedures*⟩+≡
```
    elemental function particle_data_get_charge (prt) result (charge)
      real(default) :: charge
      type(particle_data_t), intent(in) :: prt
      if (prt%charge_type /= 0) then
         charge = real (sign ((abs(prt%charge_type) - 1), &
               prt%charge_type), default) / 3
      else
         charge = 0
      end if
    end function particle_data_get_charge

    elemental function particle_data_get_mass (prt) result (mass)
      real(default) :: mass
      type(particle_data_t), intent(in) :: prt
      if (associated (prt%mass_val)) then
         mass = abs (prt%mass_val)
      else
         mass = 0
      end if
    end function particle_data_get_mass

    elemental function particle_data_get_mass_sign (prt) result (sgn)
      integer :: sgn
      type(particle_data_t), intent(in) :: prt
      if (associated (prt%mass_val)) then
         sgn = sign (1._default, prt%mass_val)
      else
         sgn = 0
      end if
    end function particle_data_get_mass_sign

    elemental function particle_data_get_width (prt) result (width)
      real(default) :: width
      type(particle_data_t), intent(in) :: prt
      if (associated (prt%width_val)) then
         width = prt%width_val
      else
         width = 0
      end if
    end function particle_data_get_width
```

```
elemental function particle_data_get_isospin (prt) result (isospin)
  real(default) :: isospin
  type(particle_data_t), intent(in) :: prt
  if (prt%isospin_type /= 0) then
     isospin = real (sign (abs(prt%isospin_type) - 1, &
           prt%isospin_type), default) / 2
  else
     isospin = 0
  end if
end function particle_data_get_isospin
```

Given an arrray of particles, return a PDG-array object that consists of all charged particles in the array

⟨*Models: procedures*⟩+≡
```
function particle_data_get_charged_pdg (prt) result (aval)
  type(pdg_array_t) :: aval
  type(particle_data_t), dimension(:), intent(in) :: prt
  aval = pack (prt%pdg, abs (prt%charge_type) > 1)
end function particle_data_get_charged_pdg
```

The same for color.

⟨*Models: procedures*⟩+≡
```
function particle_data_get_colored_pdg (prt) result (aval)
  type(pdg_array_t) :: aval
  type(particle_data_t), dimension(:), intent(in) :: prt
  aval = pack (prt%pdg, abs (prt%color_type) > 1)
end function particle_data_get_colored_pdg
```

6.1.5 Vertex data

The vertex object contains an array of particle-data pointers, for which we need a separate type. (We could use the flavor type defined in another module.)

The program does not (yet?) make use of vertex definitions, so they are not stored here.

⟨*Models: types*⟩+≡
```
type :: particle_p
   type(particle_data_t), pointer :: p => null ()
end type particle_p
```

⟨*Models: types*⟩+≡
```
type :: vertex_t
   logical :: trilinear
   integer, dimension(:), allocatable :: pdg
   type(particle_p), dimension(:), allocatable :: prt
end type vertex_t
```

Initialize using PDG codes. The model is used for finding particle data pointers associated with the pdg codes.

⟨*Models: procedures*⟩+≡

```
    subroutine vertex_init (vtx, pdg, model)
      type(vertex_t), intent(out) :: vtx
      integer, dimension(:), intent(in) :: pdg
      type(model_t), intent(in), target, optional :: model
      integer :: i
      allocate (vtx%pdg (size (pdg)))
      allocate (vtx%prt (size (pdg)))
      vtx%trilinear = size (pdg) == 3
      vtx%pdg = pdg
      if (present (model)) then
         do i = 1, size (pdg)
            vtx%prt(i)%p => model_get_particle_ptr (model, pdg(i))
         end do
      end if
    end subroutine vertex_init
```

⟨*Models: procedures*⟩+≡
```
    subroutine vertex_write (vtx, unit)
      type(vertex_t), intent(in) :: vtx
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(3x,A)", advance="no")  "vertex"
      do i = 1, size (vtx%prt)
         if (associated (vtx%prt(i)%p)) then
            write (u, "(1x,A)", advance="no") &
                 '"' // char (particle_data_get_name &
                              (vtx%prt(i)%p, vtx%pdg(i) < 0)) &
                    // '"'
         else
            write (u, "(1x,I7)", advance="no") vtx%pdg(i)
         end if
      end do
      write (u, *)
    end subroutine vertex_write
```

6.1.6 Vertex lookup table

The vertex lookup table is a hash table: given two particle codes, we check which codes are allowed for the third one.

The size of the hash table should be large enough that collisions are rare. We first select a size based on the number of vertices (multiplied by six because all permutations count), with some margin, and then choose the smallest integer power of two larger than this.

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: VERTEX_TABLE_SCALE_FACTOR = 60
```

⟨*Models: procedures*⟩+≡
```
    function vertex_table_size (n_vtx) result (n)
      integer(i32) :: n
      integer, intent(in) :: n_vtx
      integer :: i, s
```

```
      s = VERTEX_TABLE_SCALE_FACTOR * n_vtx
      n = 1
      do i = 1, 31
         n = ishft (n, 1)
         s = ishft (s,-1)
         if (s == 0)  exit
      end do
   end function vertex_table_size
```

The specific hash function takes two particle codes (arbitrary integers) and
returns a 32-bit integer. It makes use of the universal function `hash` which
operates on a byte array.

⟨*Models: procedures*⟩+≡
```
   function hash2 (pdg1, pdg2)
     integer(i32) :: hash2
     integer, intent(in) :: pdg1, pdg2
     integer(i8), dimension(1) :: mold
     hash2 = hash (transfer ((/pdg1, pdg2/), mold))
   end function hash2
```

Each entry in the vertex table stores the two particle codes and an array of
possibilities for the third code.

⟨*Models: types*⟩+≡
```
   type :: vertex_table_entry_t
      integer :: pdg1 = 0, pdg2 = 0
      integer :: n = 0
      integer, dimension(:), allocatable :: pdg3
   end type vertex_table_entry_t
```

The vertex table:

⟨*Models: types*⟩+≡
```
   type :: vertex_table_t
      type(vertex_table_entry_t), dimension(:), allocatable :: entry
      integer :: n_collisions = 0
      integer(i32) :: mask
   end type vertex_table_t
```

Initializing the vertex table: This is done in two passes. First, we scan all
permutations for all vertices and count the number of entries in each bucket of
the hashtable. Then, the buckets are allocated accordingly and filled.

Collision resolution is done by just incrementing the hash value until an
empty bucket is found. The vertex table size is fixed, since we know from the
beginning the number of entries.

⟨*Models: procedures*⟩+≡
```
   subroutine vertex_table_init (vt, prt, vtx)
     type(vertex_table_t), intent(out) :: vt
     type(particle_data_t), dimension(:), intent(in) :: prt
     type(vertex_t), dimension(:), intent(in) :: vtx
     integer :: n_prt, n_vtx, vt_size, i, p1, p2, p3
     integer, dimension(3) :: p
     n_prt = size (prt)
```

```
n_vtx = size (vtx)
vt_size = vertex_table_size (count (vtx%trilinear))
vt%mask = vt_size - 1
allocate (vt%entry (0:vt_size-1))
do i = 1, n_vtx
   if (vtx(i)%trilinear) then
      p = vtx(i)%pdg
      p1 = p(1);  p2 = p(2)
      call create (hash2 (p1, p2))
      if (p(2) /= p(3)) then
         p2 = p(3)
         call create (hash2 (p1, p2))
      end if
      if (p(1) /= p(2)) then
         p1 = p(2);  p2 = p(1)
         call create (hash2 (p1, p2))
         if (p(1) /= p(3)) then
            p2 = p(3)
            call create (hash2 (p1, p2))
         end if
      end if
      if (p(1) /= p(3)) then
         p1 = p(3);  p2 = p(1)
         call create (hash2 (p1, p2))
         if (p(1) /= p(2)) then
            p2 = p(2)
            call create (hash2 (p1, p2))
         end if
      end if
   end if
end do
do i = 0, vt_size - 1
   allocate (vt%entry(i)%pdg3 (vt%entry(i)%n))
end do
vt%entry%n = 0
do i = 1, n_vtx
   if (vtx(i)%trilinear) then
      p = vtx(i)%pdg
      p1 = p(1);  p2 = p(2);  p3 = p(3)
      call register (hash2 (p1, p2))
      if (p(2) /= p(3)) then
         p2 = p(3);  p3 = p(2)
         call register (hash2 (p1, p2))
      end if
      if (p(1) /= p(2)) then
         p1 = p(2);  p2 = p(1);  p3 = p(3)
         call register (hash2 (p1, p2))
         if (p(1) /= p(3)) then
            p2 = p(3);  p3 = p(1)
            call register (hash2 (p1, p2))
         end if
      end if
      if (p(1) /= p(3)) then
         p1 = p(3);  p2 = p(1);  p3 = p(2)
```

471

```
               call register (hash2 (p1, p2))
               if (p(1) /= p(2)) then
                   p2 = p(2);   p3 = p(1)
                   call register (hash2 (p1, p2))
               end if
            end if
        end if
      end do
   contains
     recursive subroutine create (hashval)
       integer(i32), intent(in) :: hashval
       integer :: h
       h = iand (hashval, vt%mask)
       if (vt%entry(h)%n == 0) then
          vt%entry(h)%pdg1 = p1
          vt%entry(h)%pdg2 = p2
          vt%entry(h)%n = 1
       else if (vt%entry(h)%pdg1 == p1 .and. vt%entry(h)%pdg2 == p2) then
          vt%entry(h)%n = vt%entry(h)%n + 1
       else
          vt%n_collisions = vt%n_collisions + 1
          call create (hashval + 1)
       end if
     end subroutine create
     recursive subroutine register (hashval)
       integer(i32), intent(in) :: hashval
       integer :: h
       h = iand (hashval, vt%mask)
       if (vt%entry(h)%pdg1 == p1 .and. vt%entry(h)%pdg2 == p2) then
          vt%entry(h)%n = vt%entry(h)%n + 1
          vt%entry(h)%pdg3(vt%entry(h)%n) = p3
       else
          call register (hashval + 1)
       end if
     end subroutine register
   end subroutine vertex_table_init
```

Output

⟨*Models: procedures*⟩+≡

```
   subroutine vertex_table_write (vt, unit)
     type(vertex_table_t), intent(in) :: vt
     integer, intent(in), optional :: unit
     integer :: u, i
     u = output_unit (unit);  if (u < 0)  return
     write (u, *) "vertex hash table:"
     write (u, *) "  size = ", size (vt%entry)
     write (u, *) "  used = ", count (vt%entry%n /= 0)
     write (u, *) "  coll = ", vt%n_collisions
     do i = lbound (vt%entry, 1), ubound (vt%entry, 1)
        if (vt%entry(i)%n /= 0) then
           write (u, *)  "  ", i, ":", &
                 vt%entry(i)%pdg1, vt%entry(i)%pdg2, "->", vt%entry(i)%pdg3
        end if
     end do
```

```
    end subroutine vertex_table_write
```

Return the array of particle codes that match the given pair.

⟨*Models: procedures*⟩+≡
```
    subroutine vertex_table_match (vt, pdg1, pdg2, pdg3)
      type(vertex_table_t), intent(in) :: vt
      integer, intent(in) :: pdg1, pdg2
      integer, dimension(:), allocatable, intent(out) :: pdg3
      integer :: vt_size
      vt_size = size (vt%entry)
      call match (hash2 (pdg1, pdg2))
    contains
      recursive subroutine match (hashval)
        integer(i32), intent(in) :: hashval
        integer :: h
        h = iand (hashval, vt%mask)
        if (vt%entry(h)%n == 0) then
           allocate (pdg3 (0))
        else if (vt%entry(h)%pdg1 == pdg1 .and. vt%entry(h)%pdg2 == pdg2) then
           allocate (pdg3 (size (vt%entry(h)%pdg3)))
           pdg3 = vt%entry(h)%pdg3
        else
           call match (hashval + 1)
        end if
      end subroutine match
    end subroutine vertex_table_match
```

Return true if the triplet is represented as a vertex.

⟨*Models: procedures*⟩+≡
```
    function vertex_table_check (vt, pdg1, pdg2, pdg3) result (flag)
      type(vertex_table_t), intent(in) :: vt
      integer, intent(in) :: pdg1, pdg2, pdg3
      logical :: flag
      integer :: vt_size
      vt_size = size (vt%entry)
      flag = check (hash2 (pdg1, pdg2))
    contains
      recursive function check (hashval) result (flag)
        integer(i32), intent(in) :: hashval
        integer :: h
        logical :: flag
        h = iand (hashval, vt%mask)
        if (vt%entry(h)%n == 0) then
           flag = .false.
        else if (vt%entry(h)%pdg1 == pdg1 .and. vt%entry(h)%pdg2 == pdg2) then
           flag = any (vt%entry(h)%pdg3 == pdg3)
        else
           flag = check (hashval + 1)
        end if
      end function check
    end function vertex_table_check
```

6.1.7 Model data

A model object holds all information about parameters, particles, and vertices. For models that require an external program for parameter calculation, there is the pointer to a function that does this calculation, given the set of independent and derived parameters.

⟨*Models: public*⟩+≡
```
    public :: model_t
```

⟨*Models: types*⟩+≡
```
    type :: model_t
       private
       type(string_t) :: name
       character(32) :: md5sum = ""
       type(parameter_t), dimension(:), allocatable :: par
       type(particle_data_t), dimension(:), allocatable :: prt
       type(vertex_t), dimension(:), allocatable :: vtx
       type(vertex_table_t) :: vt
       type(var_list_t) :: var_list
       type(string_t) :: dlname
       procedure(model_init_external_parameters), nopass, pointer :: &
            init_external_parameters => null ()
       type(dlaccess_t) :: dlaccess
    end type model_t
```

This is the interface for a procedure that initializes the calculation of external parameters, given the array of all parameters.

⟨*Models: interfaces*⟩≡
```
    abstract interface
       subroutine model_init_external_parameters (par) bind (C)
         import
         real(c_default_float), dimension(*), intent(inout) :: par
       end subroutine model_init_external_parameters
    end interface
```

Initialization: Specify the number of parameters, particles, vertices and allocate memory. If an associated DL library is specified, load this library.

⟨*Models: procedures*⟩+≡
```
    subroutine model_init (model, name, libname, os_data, n_par, n_prt, n_vtx)
      type(model_t), intent(out) :: model
      type(string_t), intent(in) :: name, libname
      type(os_data_t), intent(in) :: os_data
      integer, intent(in) :: n_par, n_prt, n_vtx
      type(c_funptr) :: c_fptr
      type(string_t) :: libpath
      model%name = name
      allocate (model%par (n_par))
      allocate (model%prt (n_prt))
      allocate (model%vtx (n_vtx))
      if (libname /= "") then
         if (.not. os_data%use_testfiles) then
            libpath = os_data%whizard_models_libpath_local
            model%dlname = os_get_dlname ( &
```

```
                libpath // "/" // libname, os_data, ignore=.true., silent=.true.)
           end if
           if (model%dlname == "") then
              libpath = os_data%whizard_models_libpath
              model%dlname = os_get_dlname (libpath // "/" // libname, os_data)
           end if
        else
           model%dlname = ""
        end if
        if (model%dlname /= "") then
           if (.not. dlaccess_is_open (model%dlaccess)) then
              call msg_message ("Loading model auxiliary library '" &
                   // char (libpath) // "/" // char (model%dlname) // "'")
              call dlaccess_init (model%dlaccess, os_data%whizard_models_libpath, &
                   model%dlname, os_data)
              if (dlaccess_has_error (model%dlaccess)) then
                 call msg_message (char (dlaccess_get_error (model%dlaccess)))
                 call msg_fatal ("Loading model auxiliary library '" &
                      // char (model%dlname) // "' failed")
                 return
              end if
              c_fptr = dlaccess_get_c_funptr (model%dlaccess, &
                   var_str ("init_external_parameters"))
              if (dlaccess_has_error (model%dlaccess)) then
                 call msg_message (char (dlaccess_get_error (model%dlaccess)))
                 call msg_fatal ("Loading function from auxiliary library '" &
                      // char (model%dlname) // "' failed")
                 return
              end if
              call c_f_procpointer (c_fptr, model% init_external_parameters)
           end if
        end if
     end subroutine model_init
```

Finalization: The variable list is the only part that contains pointers.

⟨*Models: procedures*⟩+≡
```
    subroutine model_final (model)
      type(model_t), intent(inout) :: model
      call var_list_final (model%var_list)
      if (model%dlname /= "")  call dlaccess_final (model%dlaccess)
    end subroutine model_final
```

Output. By default, the output is in the form of an input file. If `verbose` is true, for each derived parameter the definition (eval tree) is displayed, and the vertex hash table is shown.

⟨*Models: public*⟩+≡
```
    public :: model_write
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_write (model, unit, verbose)
      type(model_t), intent(in) :: model
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
```

```
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, *) 'model "', char (model%name), '"'
      write (u, *) '! md5sum = "', model%md5sum, '"'
      do i = 1, size (model%par)
         call parameter_write (model%par(i), unit, verbose)
         write (u, *)
      end do
      do i = 1, size (model%prt)
         call particle_data_write (model%prt(i), unit)
      end do
      do i = 1, size (model%vtx)
         call vertex_write (model%vtx(i), unit)
      end do
      if (present (verbose)) then
         if (verbose) then
            call vertex_table_write (model%vt, unit)
            call var_list_write (model%var_list, unit)
         end if
      end if
    end subroutine model_write
```

Accessing contents:

⟨*Models: public*⟩+≡
```
    public :: model_get_name
```

⟨*Models: procedures*⟩+≡
```
    function model_get_name (model) result (name)
      type(string_t) :: name
      type(model_t), intent(in) :: model
      name = model%name
    end function model_get_name
```

Retrieve the MD5 sum of a model (actually, of the model file).

⟨*Models: public*⟩+≡
```
    public :: model_get_md5sum
```

⟨*Models: procedures*⟩+≡
```
    function model_get_md5sum (model) result (md5sum)
      character(32) :: md5sum
      type(model_t), intent(in) :: model
      md5sum = model%md5sum
    end function model_get_md5sum
```

Retrieve a MD5 sum for the current model parameter values. This is done by writing them to a string, using default format.

⟨*Models: public*⟩+≡
```
    public :: model_get_parameters_md5sum
```

⟨*Models: procedures*⟩+≡
```
    function model_get_parameters_md5sum (model) result (par_md5sum)
      character(32) :: par_md5sum
      type(model_t), intent(in) :: model
```

```
    real(default), dimension(:), allocatable :: par
    integer :: unit
    call model_parameters_to_array (model, par)
    unit = free_unit ()
    open (unit, status="scratch", action="readwrite")
    write (unit, *)  par
    rewind (unit)
    par_md5sum = md5sum (unit)
    close (unit)
  end function model_get_parameters_md5sum
```

Parameters are defined by an expression which may be constant or arbitrary.

⟨*Models: interfaces*⟩+≡
```
    interface model_set_parameter
      module procedure model_set_parameter_constant
      module procedure model_set_parameter_parse_node
    end interface
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_set_parameter_constant (model, i, name, value)
      type(model_t), intent(inout), target :: model
      integer, intent(in) :: i
      type(string_t), intent(in) :: name
      real(default), intent(in) :: value
      logical, save, target :: known = .true.
      call parameter_init_independent_value (model%par(i), name, value)
      call var_list_append_real_ptr &
           (model%var_list, name, parameter_get_value_ptr (model%par(i)), known, &
           intrinsic=.true.)
    end subroutine model_set_parameter_constant

    subroutine model_set_parameter_parse_node (model, i, name, pn, constant)
      type(model_t), intent(inout), target :: model
      integer, intent(in) :: i
      type(string_t), intent(in) :: name
      type(parse_node_t), intent(in), target :: pn
      logical, intent(in) :: constant
      logical, save, target :: known = .true.
      if (constant) then
         call parameter_init_independent (model%par(i), name, pn)
      else
         call parameter_init_derived (model%par(i), name, pn, model%var_list)
      end if
      call var_list_append_real_ptr &
           (model%var_list, name, parameter_get_value_ptr (model%par(i)), &
            is_known=known, locked=.not.constant, intrinsic=.true.)
    end subroutine model_set_parameter_parse_node

    subroutine model_set_parameter_external (model, i, name)
      type(model_t), intent(inout), target :: model
      integer, intent(in) :: i
      type(string_t), intent(in) :: name
      logical, save, target :: known = .true.
```

```
      call parameter_init_external (model%par(i), name)
      call var_list_append_real_ptr &
           (model%var_list, name, parameter_get_value_ptr (model%par(i)), &
            is_known=known, locked=.true., intrinsic=.true.)
    end subroutine model_set_parameter_external
```

Return the pointer to a parameter

⟨*Models: procedures*⟩+≡
```
    function model_get_parameter_ptr (model, par_name) result (par)
      type(parameter_t), pointer :: par
      type(model_t), intent(in), target :: model
      type(string_t), intent(in), optional :: par_name
      integer :: i
      par => null ()
      if (present (par_name)) then
         do i = 1, size (model%par)
            if (model%par(i)%name == par_name) then
               par => model%par(i);  exit
            end if
         end do
         if (.not. associated (par)) then
            call msg_fatal (" Model '" // char (model%name) // "'" // &
                 " has no parameter '" // char (par_name) // "'")
         end if
      end if
    end function model_get_parameter_ptr
```

Return the value of a particular parameter

⟨*Models: public*⟩+≡
```
    public :: model_get_parameter_value
```

⟨*Models: procedures*⟩+≡
```
    function model_get_parameter_value (model, par_name) result (val)
      real(default) :: val
      type(model_t), intent(in), target :: model
      type(string_t), intent(in), optional :: par_name
      val = parameter_get_value_ptr (model_get_parameter_ptr (model, par_name))
    end function model_get_parameter_value
```

Return the number of parameters

⟨*Models: public*⟩+≡
```
    public :: model_get_n_parameters
```

⟨*Models: procedures*⟩+≡
```
    function model_get_n_parameters (model) result (n)
      integer :: n
      type(model_t), intent(in) :: model
      n = size (model%par)
    end function model_get_n_parameters
```

Transform the parameters into a single real-valued array. The first version uses
the Fortran default kind, the second one the C default kind. (Usually, the two

will be identical).

⟨*Models: public*⟩+≡
```
    public :: model_parameters_to_array
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_parameters_to_array (model, array)
      type(model_t), intent(in) :: model
      real(default), dimension(:), allocatable :: array
      integer :: i
      allocate (array (size (model%par)))
      do i = 1, size (model%par)
         array(i) = model%par(i)%value
      end do
    end subroutine model_parameters_to_array

    subroutine model_parameters_to_c_array (model, array)
      type(model_t), intent(in) :: model
      real(c_default_float), dimension(:), allocatable :: array
      allocate (array (size (model%par)))
      array = model%par%value
    end subroutine model_parameters_to_c_array

    subroutine model_parameters_from_c_array (model, array)
      type(model_t), intent(inout) :: model
      real(c_default_float), dimension(:), intent(in) :: array
      if (size (array) == size (model%par)) then
         model%par%value = array
      else
         call msg_bug ("Model '" // char (model%name) // "': size mismatch " &
              // "in parameter array")
      end if
    end subroutine model_parameters_from_c_array
```

Rcalculate all derived parameters.

⟨*Models: public*⟩+≡
```
    public :: model_parameters_update
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_parameters_update (model)
      type(model_t), intent(inout) :: model
      integer :: i
      real(default), dimension(:), allocatable :: par
      do i = 1, size (model%par)
         call parameter_reset_derived (model%par(i))
      end do
      if (associated (model% init_external_parameters)) then
         call model_parameters_to_c_array (model, par)
         call model% init_external_parameters (par)
         call model_parameters_from_c_array (model, par)
      end if
    end subroutine model_parameters_update
```

Initialize particle data with PDG long name and PDG code.

⟨*Models: procedures*⟩+≡

```
subroutine model_init_particle (model, i, longname, pdg)
  type(model_t), intent(inout) :: model
  integer, intent(in) :: i
  type(string_t), intent(in) :: longname
  integer, intent(in) :: pdg
  type(pdg_array_t) :: aval
  call particle_data_init (model%prt(i), longname, pdg)
  aval = pdg
  call var_list_append_pdg_array &
       (model%var_list, longname, aval, locked=.true., intrinsic=.true.)
end subroutine model_init_particle
```

Copy quantum numbers from another particle

⟨*Models: procedures*⟩+≡
```
subroutine model_copy_particle_data (model, i, name_src)
  type(model_t), intent(inout) :: model
  integer, intent(in) :: i
  type(string_t), intent(in) :: name_src
  call particle_data_copy (model%prt(i), &
       model_get_particle_ptr (model, &
          model_get_particle_pdg (model, name_src)))
end subroutine model_copy_particle_data
```

Set particle quantum numbers individually. Names get a ∗ prepended.

⟨*Models: procedures*⟩+≡
```
subroutine model_set_particle_data (model, i, &
     is_visible, is_parton, is_gauge, is_left_handed, is_right_handed, &
     name, anti, tex_name, tex_anti, &
     spin_type, isospin_type, charge_type, color_type, &
     mass_src, width_src)
  type(model_t), intent(inout) :: model
  integer, intent(in) :: i
  logical, intent(in), optional :: is_visible, is_parton, is_gauge
  logical, intent(in), optional :: is_left_handed, is_right_handed
  type(string_t), dimension(:), intent(in), optional :: name, anti
  type(string_t), intent(in), optional :: tex_name, tex_anti
  integer, intent(in), optional :: spin_type, isospin_type
  integer, intent(in), optional :: charge_type, color_type
  type(parameter_t), intent(in), optional, target :: mass_src, width_src
  integer :: j
  type(pdg_array_t) :: aval
  logical, parameter :: is_stable = .true.
  logical, parameter :: decays_isotropically = .false.
  logical, parameter :: decays_diagonal = .false.
  call particle_data_set (model%prt(i), &
     is_visible, is_parton, is_gauge, is_left_handed, is_right_handed, &
     is_stable, decays_isotropically, decays_diagonal, &
     is_stable, decays_isotropically, decays_diagonal, &
     name, anti, tex_name, tex_anti, &
     spin_type, isospin_type, charge_type, color_type, &
     mass_src, width_src)
  if (present (name)) then
     aval = particle_data_get_pdg (model%prt(i))
```

```
      do j = 1, size (name)
         call var_list_append_pdg_array &
              (model%var_list, name(j), aval, locked=.true., intrinsic=.true.)
      end do
   end if
   if (present (anti)) then
      aval = - particle_data_get_pdg (model%prt(i))
      do j = 1, size (anti)
         call var_list_append_pdg_array &
              (model%var_list, anti(j), aval, locked=.true., intrinsic=.true.)
      end do
   end if
end subroutine model_set_particle_data
```

⟨*Models: procedures*⟩+≡
```
   subroutine model_freeze_particle_data (model, i)
     type(model_t), intent(inout) :: model
     integer, intent(in) :: i
     call particle_data_freeze (model%prt(i))
   end subroutine model_freeze_particle_data
```

Return a pointer to the particle-data object that belongs to the specified PDG code or name.

⟨*Models: public*⟩+≡
```
   public :: model_get_particle_ptr
```

⟨*Models: procedures*⟩+≡
```
   function model_get_particle_ptr (model, pdg) result (prt)
     type(particle_data_t), pointer :: prt
     type(model_t), intent(in), target :: model
     integer, intent(in) :: pdg
     integer :: i
     prt => null ()
     if (pdg /= UNDEFINED) then
        do i = 1, size (model%prt)
           if (model%prt(i)%pdg == abs (pdg)) then
              prt => model%prt(i);  exit
           end if
        end do
        if (.not. associated (prt)) then
           write (msg_buffer, "(1x,A,1x,I7)")  "PDG code =", pdg
           call msg_message
           call msg_fatal (" Model '" // char (model%name) // "'" // &
                " has no particle with this PDG code")
        end if
     end if
   end function model_get_particle_ptr
```

Set the value, not the pointer. If the PDG code is not valid, do nothing.

⟨*Models: public*⟩+≡
```
   public :: model_set_particle_mass
   public :: model_set_particle_width
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_set_particle_mass (model, pdg, mass)
      type(model_t), intent(inout) :: model
      integer, intent(in) :: pdg
      real(default), intent(in) :: mass
      type(particle_data_t), pointer :: prt
      prt => model_get_particle_ptr (model, pdg)
      if (associated (prt))  call particle_data_set_mass (prt, mass)
    end subroutine model_set_particle_mass

    subroutine model_set_particle_width (model, pdg, width)
      type(model_t), intent(inout) :: model
      integer, intent(in) :: pdg
      real(default), intent(in) :: width
      type(particle_data_t), pointer :: prt
      prt => model_get_particle_ptr (model, pdg)
      if (associated (prt))  call particle_data_set_width (prt, width)
    end subroutine model_set_particle_width
```

Return the PDG code that matches a particle name.

⟨*Models: public*⟩+≡
```
    public :: model_get_particle_pdg
```

⟨*Models: procedures*⟩+≡
```
    function model_get_particle_pdg (model, name) result (pdg)
      integer :: pdg
      type(model_t), intent(in), target :: model
      type(string_t), intent(in) :: name
      integer :: i
      pdg = UNDEFINED
      do i = 1, size (model%prt)
         if (model%prt(i)%longname == name) then
            pdg = particle_data_get_pdg (model%prt(i));  exit
         else if (any (model%prt(i)%name == name)) then
            pdg = particle_data_get_pdg (model%prt(i));  exit
         else if (any (model%prt(i)%anti == name)) then
            pdg = - particle_data_get_pdg (model%prt(i));  exit
         end if
      end do
      if (pdg == UNDEFINED) then
         write (msg_buffer, "(1x,A,1x,A)")  "Particle name =", char (name)
         call msg_message
         call msg_fatal (" Model '" // char (model%name) // "'" // &
               " has no particle with this name")
      end if
    end function model_get_particle_pdg
```

Return a pointer to the variable list.

⟨*Models: public*⟩+≡
```
    public :: model_get_var_list_ptr
```

⟨*Models: procedures*⟩+≡
```
    function model_get_var_list_ptr (model) result (var_list)
      type(var_list_t), pointer :: var_list
```

```
      type(model_t), intent(in), target :: model
      var_list => model%var_list
    end function model_get_var_list_ptr
```

Vertex definition.

⟨Models: interfaces⟩+≡
```
    interface model_set_vertex
       module procedure model_set_vertex_pdg
       module procedure model_set_vertex_names
    end interface
```

⟨Models: procedures⟩+≡
```
    subroutine model_set_vertex_pdg (model, i, pdg)
      type(model_t), intent(inout), target :: model
      integer, intent(in) :: i
      integer, dimension(:), intent(in) :: pdg
      call vertex_init (model%vtx(i), pdg, model)
    end subroutine model_set_vertex_pdg

    subroutine model_set_vertex_names (model, i, name)
      type(model_t), intent(inout), target :: model
      integer, intent(in) :: i
      type(string_t), dimension(:), intent(in) :: name
      integer, dimension(size(name)) :: pdg
      integer :: j
      do j = 1, size (name)
         pdg(j) = model_get_particle_pdg (model, name(j))
      end do
      call vertex_init (model%vtx(i), pdg, model)
    end subroutine model_set_vertex_names
```

Lookup functions

⟨Models: public⟩+≡
```
    public :: model_match_vertex
```

⟨Models: procedures⟩+≡
```
    subroutine model_match_vertex (model, pdg1, pdg2, pdg3)
      type(model_t), intent(in) :: model
      integer, intent(in) :: pdg1, pdg2
      integer, dimension(:), allocatable, intent(out) :: pdg3
      call vertex_table_match (model%vt, pdg1, pdg2, pdg3)
    end subroutine model_match_vertex
```

⟨Models: public⟩+≡
```
    public :: model_check_vertex
```

⟨Models: procedures⟩+≡
```
    function model_check_vertex (model, pdg1, pdg2, pdg3) result (flag)
      logical :: flag
      type(model_t), intent(in) :: model
      integer, intent(in) :: pdg1, pdg2, pdg3
      flag = vertex_table_check (model%vt, pdg1, pdg2, pdg3)
    end function model_check_vertex
```

6.1.8 Reading models from file

This procedure defines the model-file syntax for the parser, returning an internal file (ifile).

Note that arithmetic operators are defined as keywords in the expression syntax, so we exclude them here.

⟨*Models: procedures*⟩+≡

```
subroutine define_model_file_syntax (ifile)
  type(ifile_t), intent(inout) :: ifile
  call ifile_append (ifile, "SEQ model_def = model_name_def " // &
       "parameters derived_pars external_pars particles vertices")
  call ifile_append (ifile, "SEQ model_name_def = model model_name")
  call ifile_append (ifile, "KEY model")
  call ifile_append (ifile, "QUO model_name = '""'...'""'")
  call ifile_append (ifile, "SEQ parameters = parameter_def*")
  call ifile_append (ifile, "SEQ parameter_def = parameter par_name " // &
       "'=' any_real_value")
  call ifile_append (ifile, "ALT any_real_value = " &
       // "neg_real_value | pos_real_value | real_value")
  call ifile_append (ifile, "SEQ neg_real_value = '-' real_value")
  call ifile_append (ifile, "SEQ pos_real_value = '+' real_value")
  call ifile_append (ifile, "KEY parameter")
  call ifile_append (ifile, "IDE par_name")
! call ifile_append (ifile, "KEY '='")
! call ifile_append (ifile, "REA par_value")
  call ifile_append (ifile, "SEQ derived_pars = derived_def*")
  call ifile_append (ifile, "SEQ derived_def = derived par_name " // &
       "'=' expr")
  call ifile_append (ifile, "KEY derived")
  call ifile_append (ifile, "SEQ external_pars = external_def*")
  call ifile_append (ifile, "SEQ external_def = external par_name")
  call ifile_append (ifile, "KEY external")
  call ifile_append (ifile, "SEQ particles = particle_def*")
  call ifile_append (ifile, "SEQ particle_def = particle prt_longname " // &
       "prt_pdg prt_details")
  call ifile_append (ifile, "KEY particle")
  call ifile_append (ifile, "IDE prt_longname")
  call ifile_append (ifile, "INT prt_pdg")
  call ifile_append (ifile, "ALT prt_details = prt_src | prt_properties")
  call ifile_append (ifile, "SEQ prt_src = like prt_longname prt_properties")
  call ifile_append (ifile, "KEY like")
  call ifile_append (ifile, "SEQ prt_properties = prt_property*")
  call ifile_append (ifile, "ALT prt_property = " // &
       "parton | invisible | gauge | left | right | " // &
       "prt_name | prt_anti | prt_tex_name | prt_tex_anti | " // &
       "prt_spin | prt_isospin | prt_charge | " // &
       "prt_color | prt_mass | prt_width")
  call ifile_append (ifile, "KEY parton")
  call ifile_append (ifile, "KEY invisible")
  call ifile_append (ifile, "KEY gauge")
  call ifile_append (ifile, "KEY left")
  call ifile_append (ifile, "KEY right")
  call ifile_append (ifile, "SEQ prt_name = name name_def+")
  call ifile_append (ifile, "SEQ prt_anti = anti name_def+")
```

```
        call ifile_append (ifile, "SEQ prt_tex_name = tex_name name_def")
        call ifile_append (ifile, "SEQ prt_tex_anti = tex_anti name_def")
        call ifile_append (ifile, "KEY name")
        call ifile_append (ifile, "KEY anti")
        call ifile_append (ifile, "KEY tex_name")
        call ifile_append (ifile, "KEY tex_anti")
        call ifile_append (ifile, "ALT name_def = name_string | name_id")
        call ifile_append (ifile, "QUO name_string = '""'...'""'")
        call ifile_append (ifile, "IDE name_id")
        call ifile_append (ifile, "SEQ prt_spin = spin frac")
        call ifile_append (ifile, "KEY spin")
!       call ifile_append (ifile, "SEQ frac = signed_int div?")
!       call ifile_append (ifile, "ALT signed_int = " &
!            // "neg_int | pos_int | integer_literal")
!       call ifile_append (ifile, "SEQ neg_int = '-' integer_literal")
!       call ifile_append (ifile, "SEQ pos_int = '+' integer_literal")
!     call ifile_append (ifile, "KEY '-'")
!     call ifile_append (ifile, "KEY '+'")
!     call ifile_append (ifile, "INT int")
!       call ifile_append (ifile, "SEQ div = '/' integer_literal")
!     call ifile_append (ifile, "KEY '/'")
        call ifile_append (ifile, "SEQ prt_isospin = isospin frac")
        call ifile_append (ifile, "KEY isospin")
        call ifile_append (ifile, "SEQ prt_charge = charge frac")
        call ifile_append (ifile, "KEY charge")
        call ifile_append (ifile, "SEQ prt_color = color integer_literal")
        call ifile_append (ifile, "KEY color")
        call ifile_append (ifile, "SEQ prt_mass = mass par_name")
        call ifile_append (ifile, "KEY mass")
        call ifile_append (ifile, "SEQ prt_width = width par_name")
        call ifile_append (ifile, "KEY width")
        call ifile_append (ifile, "SEQ vertices = vertex_def*")
        call ifile_append (ifile, "SEQ vertex_def = vertex name_def+")
        call ifile_append (ifile, "KEY vertex")
        call define_expr_syntax (ifile, particles=.false., analysis=.false.)
      end subroutine define_model_file_syntax
```

The model-file syntax and lexer are fixed, therefore stored as module variables:

⟨*Models: variables*⟩≡
```
      type(syntax_t), target, save :: syntax_model_file
```

⟨*Models: public*⟩+≡
```
      public :: syntax_model_file_init
```

⟨*Models: procedures*⟩+≡
```
      subroutine syntax_model_file_init ()
        type(ifile_t) :: ifile
        call define_model_file_syntax (ifile)
        call syntax_init (syntax_model_file, ifile)
        call ifile_final (ifile)
      end subroutine syntax_model_file_init
```

⟨*Models: procedures*⟩+≡

```
    subroutine lexer_init_model_file (lexer)
      type(lexer_t), intent(out) :: lexer
      call lexer_init (lexer, &
           comment_chars = "#!", &
           quote_chars = '"{', &
           quote_match = '"}', &
           single_chars = ":()", &
           special_class = (/ "+-*/^", "<>=  " /) , &
           keyword_list = syntax_get_keyword_list_ptr (syntax_model_file))
    end subroutine lexer_init_model_file
```

⟨*Models: public*⟩+≡
```
    public :: syntax_model_file_final
```

⟨*Models: procedures*⟩+≡
```
    subroutine syntax_model_file_final ()
      call syntax_final (syntax_model_file)
    end subroutine syntax_model_file_final
```

⟨*Models: public*⟩+≡
```
    public :: syntax_model_file_write
```

⟨*Models: procedures*⟩+≡
```
    subroutine syntax_model_file_write (unit)
      integer, intent(in), optional :: unit
      call syntax_write (syntax_model_file, unit)
    end subroutine syntax_model_file_write
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_read (model, filename, os_data, exist)
      type(model_t), intent(out), target :: model
      type(string_t), intent(in) :: filename
      type(os_data_t), intent(in) :: os_data
      logical, intent(out) :: exist
      type(string_t) :: file
      type(stream_t), target :: stream
      type(lexer_t) :: lexer
      integer :: unit
      character(32) :: model_md5sum
      type(parse_tree_t) :: parse_tree
      type(parse_node_t), pointer :: nd_model_def, nd_model_name_def
      type(parse_node_t), pointer :: nd_model_arg
      type(parse_node_t), pointer :: nd_parameters, nd_derived_pars
      type(parse_node_t), pointer :: nd_external_pars
      type(parse_node_t), pointer :: nd_particles, nd_vertices
      type(string_t) :: model_name, lib_name
      integer :: n_par, n_der, n_ext, n_prt, n_vtx
      real(c_default_float), dimension(:), allocatable :: par
      integer :: i
      type(parse_node_t), pointer :: nd_par_def
      type(parse_node_t), pointer :: nd_der_def
      type(parse_node_t), pointer :: nd_ext_def
      type(parse_node_t), pointer :: nd_prt
      type(parse_node_t), pointer :: nd_vtx
```

```fortran
    type(pdg_array_t) :: prt_undefined
    file = filename
    inquire (file=char(file), exist=exist)
    if ((.not. exist) .and. (.not. os_data%use_testfiles)) then
       file = os_data%whizard_modelpath_local // "/" // filename
       inquire (file = char (file), exist = exist)
    end if
    if (.not. exist) then
       file = os_data%whizard_modelpath // "/" // filename
       inquire (file = char (file), exist = exist)
    end if
    if (.not. exist) then
       call msg_fatal ("Model file '" // char (filename) // "' not found")
       return
    end if
    call msg_message ("Reading model file '" // char (file) // "'")
    call lexer_init_model_file (lexer)
    unit = free_unit ()
    open (file=char(file), unit=unit, action="read", status="old")
    model_md5sum = md5sum (unit)
    close (unit)
    call stream_init (stream, char (file))
    call lexer_assign_stream (lexer, stream)
    call parse_tree_init (parse_tree, syntax_model_file, lexer)
    call stream_final (stream)
    call lexer_final (lexer)
!     call parse_tree_write (parse_tree)
    nd_model_def => parse_tree_get_root_ptr (parse_tree)
    nd_model_name_def => parse_node_get_sub_ptr (nd_model_def)
    model_name = parse_node_get_string &
         (parse_node_get_sub_ptr (nd_model_name_def, 2))
    nd_parameters => parse_node_get_next_ptr (nd_model_name_def)
    if (associated (nd_parameters)) then
       if (parse_node_get_rule_key (nd_parameters) == "parameters") then
          n_par = parse_node_get_n_sub (nd_parameters)
          nd_par_def => parse_node_get_sub_ptr (nd_parameters)
          nd_derived_pars => parse_node_get_next_ptr (nd_parameters)
       else
          n_par = 0
          nd_derived_pars => nd_parameters
          nd_parameters => null ()
       end if
    else
       n_par = 0
       nd_derived_pars => null ()
    end if
    if (associated (nd_derived_pars)) then
       if (parse_node_get_rule_key (nd_derived_pars) == "derived_pars") then
          n_der = parse_node_get_n_sub (nd_derived_pars)
          nd_der_def => parse_node_get_sub_ptr (nd_derived_pars)
          nd_external_pars => parse_node_get_next_ptr (nd_derived_pars)
       else
          n_der = 0
          nd_external_pars => nd_derived_pars
```

```
            nd_derived_pars => null ()
         end if
      else
         n_der = 0
         nd_external_pars => null ()
      end if
      if (associated (nd_external_pars)) then
         if (parse_node_get_rule_key (nd_external_pars) == "external_pars") then
            n_ext = parse_node_get_n_sub (nd_external_pars)
            lib_name = "external." // model_name
            nd_ext_def => parse_node_get_sub_ptr (nd_external_pars)
            nd_particles => parse_node_get_next_ptr (nd_external_pars)
         else
            n_ext = 0
            lib_name = ""
            nd_particles => nd_external_pars
            nd_external_pars => null ()
         end if
      else
         n_ext = 0
         lib_name = ""
         nd_particles => null ()
      end if
      if (associated (nd_particles)) then
         if (parse_node_get_rule_key (nd_particles) == "particles") then
            n_prt = parse_node_get_n_sub (nd_particles)
            nd_prt => parse_node_get_sub_ptr (nd_particles)
            nd_vertices => parse_node_get_next_ptr (nd_particles)
         else
            n_prt = 0
            nd_vertices => nd_particles
            nd_particles => null ()
         end if
      else
         n_prt = 0
         nd_vertices => null ()
      end if
      if (associated (nd_vertices)) then
         n_vtx = parse_node_get_n_sub (nd_vertices)
         nd_vtx => parse_node_get_sub_ptr (nd_vertices)
      else
         n_vtx = 0
      end if
      call model_init (model, model_name, lib_name, os_data, &
           n_par + n_der + n_ext, n_prt, n_vtx)
      model%md5sum = model_md5sum
      do i = 1, n_par
         call model_read_parameter (model, i, nd_par_def)
         nd_par_def => parse_node_get_next_ptr (nd_par_def)
      end do
      do i = n_par + 1, n_par + n_der
         call model_read_derived (model, i, nd_der_def)
         nd_der_def => parse_node_get_next_ptr (nd_der_def)
      end do
```

```
      do i = n_par + n_der + 1, n_par + n_der + n_ext
         call model_read_external (model, i, nd_ext_def)
         nd_ext_def => parse_node_get_next_ptr (nd_ext_def)
      end do
      if (associated (model% init_external_parameters)) then
         call model_parameters_to_c_array (model, par)
         call model% init_external_parameters (par)
         call model_parameters_from_c_array (model, par)
      end if
      prt_undefined = UNDEFINED
      call var_list_append_pdg_array &
           (model%var_list, var_str ("particle"), &
            prt_undefined, locked = .true., intrinsic=.true.)
      do i = 1, n_prt
         call model_read_particle (model, i, nd_prt)
         nd_prt => parse_node_get_next_ptr (nd_prt)
      end do
      do i = 1, n_vtx
         call model_read_vertex (model, i, nd_vtx)
         nd_vtx => parse_node_get_next_ptr (nd_vtx)
      end do
      call parse_tree_final (parse_tree)
      call var_list_append_pdg_array &
           (model%var_list, var_str ("charged"), &
            particle_data_get_charged_pdg (model%prt), locked = .true., &
            intrinsic=.true.)
      call var_list_append_pdg_array &
           (model%var_list, var_str ("colored"), &
            particle_data_get_colored_pdg (model%prt), locked = .true., &
            intrinsic=.true.)
    end subroutine model_read
```

Parameters are real values (literal) with an optional unit.

⟨*Models: procedures*⟩+≡

```
    subroutine model_read_parameter (model, i, node)
      type(model_t), intent(inout), target :: model
      integer, intent(in) :: i
      type(parse_node_t), intent(in), target :: node
      type(parse_node_t), pointer :: node_name, node_val
      type(string_t) :: name
      node_name => parse_node_get_sub_ptr (node, 2)
      name = parse_node_get_string (node_name)
      node_val => parse_node_get_next_ptr (node_name, 2)
      call model_set_parameter (model, i, name, node_val, constant=.true.)
    end subroutine model_read_parameter
```

Derived parameters have any numeric expression as their definition.

⟨*Models: procedures*⟩+≡

```
    subroutine model_read_derived (model, i, node)
      type(model_t), intent(inout), target :: model
      integer, intent(in) :: i
      type(parse_node_t), intent(in), target :: node
      type(string_t) :: name
```

```
    type(parse_node_t), pointer :: pn_expr
    name = parse_node_get_string (parse_node_get_sub_ptr (node, 2))
    pn_expr => parse_node_get_sub_ptr (node, 4)
    call model_set_parameter (model, i, name, pn_expr, constant=.false.)
  end subroutine model_read_derived
```

External parameters have no definition; they are handled by an external library.

⟨*Models: procedures*⟩+≡

```
  subroutine model_read_external (model, i, node)
    type(model_t), intent(inout), target :: model
    integer, intent(in) :: i
    type(parse_node_t), intent(in), target :: node
    type(string_t) :: name
    name = parse_node_get_string (parse_node_get_sub_ptr (node, 2))
    call model_set_parameter_external (model, i, name)
  end subroutine model_read_external
```

⟨*Models: procedures*⟩+≡

```
  subroutine model_read_particle (model, i, node)
    type(model_t), intent(inout) :: model
    integer, intent(in) :: i
    type(parse_node_t), intent(in) :: node
    type(parse_node_t), pointer :: nd_src, nd_props, nd_prop
    type(string_t) :: longname
    integer :: pdg
    type(string_t) :: name_src
    type(string_t), dimension(:), allocatable :: name
    longname = parse_node_get_string (parse_node_get_sub_ptr (node, 2))
    pdg = parse_node_get_integer (parse_node_get_sub_ptr (node, 3))
    call model_init_particle (model, i, longname, pdg)
    nd_src => parse_node_get_sub_ptr (node, 4)
    if (associated (nd_src)) then
       if (parse_node_get_rule_key (nd_src) == "prt_src") then
          name_src = parse_node_get_string (parse_node_get_sub_ptr (nd_src, 2))
          call model_copy_particle_data (model, i, name_src)
          nd_props => parse_node_get_sub_ptr (nd_src, 3)
       else
          nd_props => nd_src
       end if
       nd_prop => parse_node_get_sub_ptr (nd_props)
       do while (associated (nd_prop))
          select case (char (parse_node_get_rule_key (nd_prop)))
          case ("invisible")
             call model_set_particle_data (model, i, is_visible=.false.)
          case ("parton")
             call model_set_particle_data (model, i, is_parton=.true.)
          case ("gauge")
             call model_set_particle_data (model, i, is_gauge=.true.)
          case ("left")
             call model_set_particle_data (model, i, is_left_handed=.true.)
          case ("right")
             call model_set_particle_data (model, i, is_right_handed=.true.)
          case ("prt_name")
```

```
                 call read_names (nd_prop, name)
                 call model_set_particle_data (model, i, name=name)
              case ("prt_anti")
                 call read_names (nd_prop, name)
                 call model_set_particle_data (model, i, anti=name)
              case ("prt_tex_name")
                 call model_set_particle_data (model, i, &
                      tex_name = parse_node_get_string &
                      (parse_node_get_sub_ptr (nd_prop, 2)))
              case ("prt_tex_anti")
                 call model_set_particle_data (model, i, &
                      tex_anti = parse_node_get_string &
                      (parse_node_get_sub_ptr (nd_prop, 2)))
              case ("prt_spin")
                 call model_set_particle_data (model, i, &
                      spin_type = read_frac &
                      (parse_node_get_sub_ptr (nd_prop, 2), 2))
              case ("prt_isospin")
                 call model_set_particle_data (model, i, &
                      isospin_type = read_frac &
                      (parse_node_get_sub_ptr (nd_prop, 2), 2))
              case ("prt_charge")
                 call model_set_particle_data (model, i, &
                      charge_type = read_frac &
                      (parse_node_get_sub_ptr (nd_prop, 2), 3))
              case ("prt_color")
                 call model_set_particle_data (model, i, &
                      color_type = parse_node_get_integer &
                      (parse_node_get_sub_ptr (nd_prop, 2)))
              case ("prt_mass")
                 call model_set_particle_data (model, i, &
                      mass_src = model_get_parameter_ptr &
                      (model, parse_node_get_string &
                      (parse_node_get_sub_ptr (nd_prop, 2))))
              case ("prt_width")
                 call model_set_particle_data (model, i, &
                      width_src = model_get_parameter_ptr &
                      (model, parse_node_get_string &
                      (parse_node_get_sub_ptr (nd_prop, 2))))
              case default
                 call msg_bug (" Unknown particle property '" &
                      // char (parse_node_get_rule_key (nd_prop)) // "'")
              end select
              if (allocated (name))  deallocate (name)
              nd_prop => parse_node_get_next_ptr (nd_prop)
           end do
        end if
        call model_freeze_particle_data (model, i)
      end subroutine model_read_particle
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_read_vertex (model, i, node)
      type(model_t), intent(inout) :: model
      integer, intent(in) :: i
```

```
      type(parse_node_t), intent(in) :: node
      type(string_t), dimension(:), allocatable :: name
      call read_names (node, name)
      call model_set_vertex (model, i, name)
    end subroutine model_read_vertex
```

⟨*Models: procedures*⟩+≡
```
    subroutine read_names (node, name)
      type(parse_node_t), intent(in) :: node
      type(string_t), dimension(:), allocatable, intent(inout) :: name
      type(parse_node_t), pointer :: nd_name
      integer :: n_names, i
      n_names = parse_node_get_n_sub (node) - 1
      allocate (name (n_names))
      nd_name => parse_node_get_sub_ptr (node, 2)
      do i = 1, n_names
         name(i) = parse_node_get_string (nd_name)
         nd_name => parse_node_get_next_ptr (nd_name)
      end do
    end subroutine read_names
```

⟨*Models: procedures*⟩+≡
```
    function read_frac (nd_frac, base) result (qn_type)
      integer :: qn_type
      type(parse_node_t), intent(in) :: nd_frac
      integer, intent(in) :: base
      type(parse_node_t), pointer :: nd_num, nd_den
      integer :: num, den
      nd_num => parse_node_get_sub_ptr (nd_frac)
      nd_den => parse_node_get_next_ptr (nd_num)
      select case (char (parse_node_get_rule_key (nd_num)))
      case ("integer_literal")
         num = parse_node_get_integer (nd_num)
      case ("neg_int")
         num = - parse_node_get_integer (parse_node_get_sub_ptr (nd_num, 2))
      case ("pos_int")
         num = parse_node_get_integer (parse_node_get_sub_ptr (nd_num, 2))
      case default
         call parse_tree_bug (nd_num, "int|neg_int|pos_int")
      end select
      if (associated (nd_den)) then
         den = parse_node_get_integer (parse_node_get_sub_ptr (nd_den, 2))
      else
         den = 1
      end if
      if (den == 1) then
         qn_type = sign (1 + abs (num) * base, num)
      else if (den == base) then
         qn_type = sign (abs (num) + 1, num)
      else
         call parse_node_write_rec (nd_frac)
         call msg_fatal (" Fractional quantum number: wrong denominator")
      end if
```

```
    end function read_frac
```

6.1.9 Model list

List of currently active models

⟨*Models: types*⟩+≡
```
    type :: model_entry_t
      type(model_t) :: model
      type(model_entry_t), pointer :: next => null ()
    end type model_entry_t
```

⟨*Models: types*⟩+≡
```
    type :: model_list_t
      type(model_entry_t), pointer :: first => null ()
      type(model_entry_t), pointer :: last => null ()
    end type model_list_t
```

The model list is stored as a module variable. Thus, the operations acting on
the list do not have the model list as an argument.

⟨*Models: variables*⟩+≡
```
    type(model_list_t), target, save :: model_list
```

Write an account of the model list.

⟨*Models: public*⟩+≡
```
    public :: model_list_write
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_list_write (unit, verbose)
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      type(model_entry_t), pointer :: current
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, *) "List of models:"
      current => model_list%first
      if (associated (current)) then
         do while (associated (current))
            write (u, *)
            call model_write (current%model, unit, verbose)
            current => current%next
         end do
      else
         write (u, *) "  [empty]"
      end if
    end subroutine model_list_write
```

Add a new model with given **name** to the list, if it does not yet exist. If successful,
return a pointer to the new model.

⟨*Models: procedures*⟩+≡
```
    subroutine model_list_add (name, os_data, n_par, n_prt, n_vtx, model)
```

```
      type(string_t), intent(in) :: name
      type(os_data_t), intent(in) :: os_data
      integer, intent(in) :: n_par, n_prt, n_vtx
      type(model_t), pointer :: model
      type(model_entry_t), pointer :: current
      if (model_list_model_exists (name)) then
         model => null ()
      else
         allocate (current)
         if (associated (model_list%first)) then
            model_list%last%next => current
         else
            model_list%first => current
         end if
         model_list%last => current
         model => current%model
         call model_init (model, name, var_str (""), os_data, &
               n_par, n_prt, n_vtx)
      end if
   end subroutine model_list_add
```

Read a new model from file and add to the list, if it does not yet exist. Finalize
the model by allocating the vertex table. Return a pointer to the new model.
If unsuccessful, return the original pointer.

⟨*Models: public*⟩+≡
```
    public :: model_list_read_model
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_list_read_model (name, filename, os_data, model)
      type(string_t), intent(in) :: name, filename
      type(os_data_t), intent(in) :: os_data
      type(model_t), pointer :: model
      type(model_entry_t), pointer :: current
      logical :: exist
      if (.not. model_list_model_exists (name)) then
         allocate (current)
         call model_read (current%model, filename, os_data, exist)
         if (.not. exist)  return
         if (current%model%name /= name) then
            call msg_fatal ("Model file '" // char (filename) // &
                  "' contains model '" // char (current%model%name) // &
                  "' instead of '" // char (name) // "'")
            call model_final (current%model);  deallocate (current)
            return
         end if
         if (associated (model_list%first)) then
            model_list%last%next => current
         else
            model_list%first => current
         end if
         model_list%last => current
         call vertex_table_init &
               (current%model%vt, current%model%prt, current%model%vtx)
         model => current%model
```

494

```
        else
            model => model_list_get_model_ptr (name)
        end if
    end subroutine model_list_read_model
```

Check if a model exists by examining the list

⟨*Models: public*⟩+≡
```
    public :: model_list_model_exists
```

⟨*Models: procedures*⟩+≡
```
    function model_list_model_exists (name) result (exists)
      logical :: exists
      type(string_t), intent(in) :: name
      type(model_entry_t), pointer :: current
      current => model_list%first
      do while (associated (current))
         if (current%model%name == name) then
            exists = .true.
            return
         end if
         current => current%next
      end do
      exists = .false.
    end function model_list_model_exists
```

Return a pointer to a named model

⟨*Models: public*⟩+≡
```
    public :: model_list_get_model_ptr
```

⟨*Models: procedures*⟩+≡
```
    function model_list_get_model_ptr (name) result (model)
      type(model_t), pointer :: model
      type(string_t), intent(in) :: name
      type(model_entry_t), pointer :: current
      current => model_list%first
      do while (associated (current))
         if (current%model%name == name) then
            model => current%model
            return
         end if
         current => current%next
      end do
      model => null ()
    end function model_list_get_model_ptr
```

Delete the list of models

⟨*Models: public*⟩+≡
```
    public :: model_list_final
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_list_final ()
      type(model_entry_t), pointer :: current
      model_list%last => null ()
      do while (associated (model_list%first))
```

```
          current => model_list%first
          model_list%first => model_list%first%next
          call model_final (current%model)
          deallocate (current)
       end do
     end subroutine model_list_final
```

6.1.10   Test

⟨*Models: public*⟩+≡
```
    public :: models_test
```

⟨*Models: procedures*⟩+≡
```
    subroutine models_test ()
      type(os_data_t) :: os_data
      call syntax_model_file_init ()
      call syntax_write (syntax_model_file)
      print *
      call models_test1 (os_data)
      call models_test2 (os_data)
      call models_test2 (os_data)
      call model_list_write (verbose=.true.)
  !   call model_list_write ()
      call model_list_final ()
      call syntax_model_file_final ()
    end subroutine models_test

    subroutine models_test1 (os_data)
      type(os_data_t), intent(in) :: os_data
      type(model_t), pointer :: model
      type(string_t) :: model_name
      type(string_t) :: x_longname
      type(string_t), dimension(2) :: parname
      type(string_t), dimension(2) :: x_name
      type(string_t), dimension(1) :: x_anti
      type(string_t) :: x_tex_name, x_tex_anti
      type(string_t) :: y_longname
      type(string_t), dimension(2) :: y_name
      type(string_t) :: y_tex_name
      model_name = "Test model"
      call model_list_add (model_name, os_data, 2, 2, 3, model)
      parname(1) = "mx"
      parname(2) = "coup"
      call model_set_parameter (model, 1, parname(1), 10._default)
      call model_set_parameter (model, 2, parname(2), 1.3_default)
      print *
      x_longname = "X_LEPTON"
      x_name(1) = "X"
      x_name(2) = "x"
      x_anti(1) = "Xbar"
      x_tex_name = "X^+"
      x_tex_anti = "X^-"
      call model_init_particle (model, 1, x_longname, 99)
```

```
      call model_set_particle_data (model, 1, &
           .true., .false., .false., .false., .false., &
           x_name, x_anti, x_tex_name, x_tex_anti, &
           SPINOR, -3, 2, 1, model_get_parameter_ptr (model, parname(1)))
    y_longname = "Y_COLORON"
    y_name(1) = "Y"
    y_name(2) = "yc"
    y_tex_name = "Y^0"
    call model_init_particle (model, 2, y_longname, 97)
    call model_set_particle_data (model, 2,  &
           .false., .false., .true., .false., .false., &
            name=y_name, tex_name=y_tex_name, &
            spin_type=SCALAR, isospin_type=2, charge_type=1, color_type=8)
    call model_set_vertex (model, 1, (/ 99, 99, 99 /))
    call model_set_vertex (model, 2, (/ 99, 99, 99, 99 /))
    call model_set_vertex (model, 3, (/ 99, 97, 99 /))
end subroutine models_test1

subroutine models_test2 (os_data)
  type(os_data_t), intent(in) :: os_data
  type(string_t) :: name, filename
  type(model_t), pointer :: model
  name = "QCD"
  filename = "test.mdl"
  call model_list_read_model (name, filename, os_data, model)
end subroutine models_test2
```

Chapter 7

Quantum Numbers

We introduce separate types and modules for particle quantum numbers and use them for defining independent particles and entangled states.

helicities Types and methods for spin density matrices.

colors Dealing with colored particles, using the color-flow representation.

flavors PDG codes and particle properties, depends on the model.

quantum_states Quantum numbers and density matrices for entangled particle systems.

7.1 Helicities

This module defines types and tools for dealing with helicity information.

⟨`helicities.f90`⟩≡
  ⟨*File header*⟩

  ```
  module helicities
  ```

  ⟨*Use file utils*⟩

  ⟨*Standard module head*⟩

  ⟨*Helicities: public*⟩

  ⟨*Helicities: types*⟩

  ⟨*Helicities: interfaces*⟩

  ```
  contains
  ```

  ⟨*Helicities: procedures*⟩

  ```
  end module helicities
  ```

7.1.1 Helicity types

Helicities may be defined or undefined, corresponding to a polarized or unpo-
larized state. Each helicity is actually a pair of helicities, corresponding to an
entry in the spin density matrix. Obviously, diagonal entries are distinguished.
In addition, we have a ghost flag that would apply to FP ghosts in particular.

⟨*Helicities: public*⟩≡
```
    public :: helicity_t
```

⟨*Helicities: types*⟩≡
```
    type :: helicity_t
      private
      logical :: defined = .false.
      integer :: h1, h2
      logical :: ghost = .false.
    end type helicity_t
```

Initializers:

⟨*Helicities: public*⟩+≡
```
    public :: helicity_init
```

⟨*Helicities: interfaces*⟩≡
```
    interface helicity_init
      module procedure helicity_init0, helicity_init0g
      module procedure helicity_init1, helicity_init1g
      module procedure helicity_init2, helicity_init2g
    end interface
```

⟨Helicities: procedures⟩≡
```fortran
    elemental subroutine helicity_init0 (hel)
      type(helicity_t), intent(out) :: hel
    end subroutine helicity_init0

    elemental subroutine helicity_init0g (hel, ghost)
      type(helicity_t), intent(out) :: hel
      logical, intent(in) :: ghost
      hel%ghost = ghost
    end subroutine helicity_init0g

    elemental subroutine helicity_init1 (hel, h)
      type(helicity_t), intent(out) :: hel
      integer, intent(in) :: h
      hel%defined = .true.
      hel%h1 = h
      hel%h2 = h
    end subroutine helicity_init1

    elemental subroutine helicity_init1g (hel, h, ghost)
      type(helicity_t), intent(out) :: hel
      integer, intent(in) :: h
      logical, intent(in) :: ghost
      call helicity_init1 (hel, h)
      hel%ghost = ghost
    end subroutine helicity_init1g

    elemental subroutine helicity_init2 (hel, h2, h1)
      type(helicity_t), intent(out) :: hel
      integer, intent(in) :: h1, h2
      hel%defined = .true.
      hel%h2 = h2
      hel%h1 = h1
    end subroutine helicity_init2

    elemental subroutine helicity_init2g (hel, h2, h1, ghost)
      type(helicity_t), intent(out) :: hel
      integer, intent(in) :: h1, h2
      logical, intent(in) :: ghost
      call helicity_init2 (hel, h2, h1)
      hel%ghost = ghost
    end subroutine helicity_init2g
```

Set the ghost property separately:

⟨Helicities: public⟩+≡
```fortran
    public :: helicity_set_ghost
```

⟨Helicities: procedures⟩+≡
```fortran
    elemental subroutine helicity_set_ghost (hel, ghost)
      type(helicity_t), intent(inout) :: hel
      logical, intent(in) :: ghost
      hel%ghost = ghost
    end subroutine helicity_set_ghost
```

Undefine:

⟨*Helicities: public*⟩+≡
```
    public :: helicity_undefine
```

⟨*Helicities: procedures*⟩+≡
```
    elemental subroutine helicity_undefine (hel)
      type(helicity_t), intent(inout) :: hel
      hel%defined = .false.
      hel%ghost = .false.
    end subroutine helicity_undefine
```

Diagonalize by removing the second entry (use with care!)

⟨*Helicities: public*⟩+≡
```
    public :: helicity_diagonalize
```

⟨*Helicities: procedures*⟩+≡
```
    elemental subroutine helicity_diagonalize (hel)
      type(helicity_t), intent(inout) :: hel
      hel%h2 = hel%h1
    end subroutine helicity_diagonalize
```

Output (no linebreak). No output if undefined.

⟨*Helicities: public*⟩+≡
```
    public :: helicity_write
```

⟨*Helicities: procedures*⟩+≡
```
    subroutine helicity_write (hel, unit)
      type(helicity_t), intent(in) :: hel
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      if (hel%defined) then
         if (hel%ghost) then
            write (u, "(A)", advance="no")  "h*("
         else
            write (u, "(A)", advance="no")  "h("
         end if
         write (u, "(I0)", advance="no")  hel%h1
         if (hel%h1 /= hel%h2) then
            write (u, "(A)", advance="no") "|"
            write (u, "(I0)", advance="no")  hel%h2
         end if
         write (u, "(A)", advance="no")  ")"
      else if (hel%ghost) then
         write (u, "(A)", advance="no")  "h*"
      end if
    end subroutine helicity_write
```

Binary I/O. Write contents only if defined.

⟨*Helicities: public*⟩+≡
```
    public :: helicity_write_raw
    public :: helicity_read_raw
```

⟨*Helicities: procedures*⟩+≡
```
    subroutine helicity_write_raw (hel, u)
      type(helicity_t), intent(in) :: hel
      integer, intent(in) :: u
      write (u) hel%defined
      if (hel%defined) then
         write (u) hel%h1, hel%h2
         write (u) hel%ghost
      end if
    end subroutine helicity_write_raw

    subroutine helicity_read_raw (hel, u, iostat)
      type(helicity_t), intent(out) :: hel
      integer, intent(in) :: u
      integer, intent(out), optional :: iostat
      read (u, iostat=iostat) hel%defined
      if (hel%defined) then
         read (u, iostat=iostat) hel%h1, hel%h2
         read (u, iostat=iostat) hel%ghost
      end if
    end subroutine helicity_read_raw
```

7.1.2   Predicates

Check if the helicity is defined:

⟨*Helicities: public*⟩+≡
```
    public :: helicity_is_defined
```
⟨*Helicities: procedures*⟩+≡
```
    elemental function helicity_is_defined (hel) result (defined)
      logical :: defined
      type(helicity_t), intent(in) :: hel
      defined = hel%defined
    end function helicity_is_defined
```

Return true if the two helicities are equal or the particle is unpolarized:

⟨*Helicities: public*⟩+≡
```
    public :: helicity_is_diagonal
```
⟨*Helicities: procedures*⟩+≡
```
    elemental function helicity_is_diagonal (hel) result (diagonal)
      logical :: diagonal
      type(helicity_t), intent(in) :: hel
      if (hel%defined) then
         diagonal = hel%h1 == hel%h2
      else
         diagonal = .true.
      end if
    end function helicity_is_diagonal
```

Return the ghost flag

⟨*Helicities: public*⟩+≡
```
    public :: helicity_is_ghost
```

⟨*Helicities: procedures*⟩+≡
```
    elemental function helicity_is_ghost (hel) result (ghost)
      logical :: ghost
      type(helicity_t), intent(in) :: hel
      ghost = hel%ghost
    end function helicity_is_ghost
```

7.1.3 Accessing contents

This returns a two-element array and thus cannot be elemental. The result is unpredictable if the helicity is undefined.

⟨*Helicities: public*⟩+≡
```
    public :: helicity_get
```

⟨*Helicities: procedures*⟩+≡
```
    pure function helicity_get (hel) result (h)
      integer, dimension(2) :: h
      type(helicity_t), intent(in) :: hel
      h(1) = hel%h2
      h(2) = hel%h1
    end function helicity_get
```

7.1.4 Comparisons

When comparing helicities, if either one is undefined, they are considered to match. In other words, an unpolarized particle matches any polarization. In the `dmatch` variant, it matches only diagonal helicity.

The ghost flag is ignored when matching, but matters when testing for equality.

⟨*Helicities: public*⟩+≡
```
    public :: operator(.match.)
    public :: operator(.dmatch.)
    public :: operator(==)
    public :: operator(/=)
```

⟨*Helicities: interfaces*⟩+≡
```
    interface operator(.match.)
       module procedure helicity_match
    end interface
    interface operator(.dmatch.)
       module procedure helicity_match_diagonal
    end interface
    interface operator(==)
       module procedure helicity_eq
    end interface
    interface operator(/=)
       module procedure helicity_neq
    end interface
```

⟨*Helicities: procedures*⟩+≡
```
elemental function helicity_match (hel1, hel2) result (eq)
  logical :: eq
  type(helicity_t), intent(in) :: hel1, hel2
  if (hel1%defined .and. hel2%defined) then
     eq = (hel1%h1 == hel2%h1) .and. (hel1%h2 == hel2%h2)
  else
     eq = .true.
  end if
end function helicity_match

elemental function helicity_match_diagonal (hel1, hel2) result (eq)
  logical :: eq
  type(helicity_t), intent(in) :: hel1, hel2
  if (hel1%defined .and. hel2%defined) then
     eq = (hel1%h1 == hel2%h1) .and. (hel1%h2 == hel2%h2)
  else if (hel1%defined) then
     eq = hel1%h1 == hel1%h2
  else if (hel2%defined) then
     eq = hel2%h1 == hel2%h2
  else
     eq = .true.
  end if
end function helicity_match_diagonal
```

⟨*Helicities: procedures*⟩+≡
```
elemental function helicity_eq (hel1, hel2) result (eq)
  logical :: eq
  type(helicity_t), intent(in) :: hel1, hel2
  if (hel1%defined .and. hel2%defined) then
     eq = (hel1%h1 == hel2%h1) .and. (hel1%h2 == hel2%h2) &
           .and. (hel1%ghost .eqv. hel2%ghost)
  else if (.not. hel1%defined .and. .not. hel2%defined) then
     eq = hel1%ghost .eqv. hel2%ghost
  else
     eq = .false.
  end if
end function helicity_eq
```

⟨*Helicities: procedures*⟩+≡
```
elemental function helicity_neq (hel1, hel2) result (neq)
  logical :: neq
  type(helicity_t), intent(in) :: hel1, hel2
  if (hel1%defined .and. hel2%defined) then
     neq = (hel1%h1 /= hel2%h1) .or. (hel1%h2 /= hel2%h2) &
           .or. (hel1%ghost .neqv. hel2%ghost)
  else if (.not. hel1%defined .and. .not. hel2%defined) then
     neq = hel1%ghost .neqv. hel2%ghost
  else
     neq = .true.
  end if
end function helicity_neq
```

7.1.5 Tools

Merge two helicity objects by taking the first entry from the first and the second entry from the second argument. Makes sense only if the input helicities were defined and diagonal. The handling of ghost flags is not well-defined; one should verify beforehand that they match.

⟨*Helicities: public*⟩+≡
```
public :: operator(.merge.)
```

⟨*Helicities: interfaces*⟩+≡
```
interface operator(.merge.)
   module procedure merge_helicities
end interface
```

⟨*Helicities: procedures*⟩+≡
```
elemental function merge_helicities (hel1, hel2) result (hel)
  type(helicity_t) :: hel
  type(helicity_t), intent(in) :: hel1, hel2
  if (helicity_is_defined (hel1) .and. helicity_is_defined (hel2)) then
     call helicity_init2g (hel, hel2%h1, hel1%h1, hel1%ghost)
  else if (helicity_is_defined (hel1)) then
     hel = hel1
  else if (helicity_is_defined (hel2)) then
     hel = hel2
  end if
end function merge_helicities
```

7.2 Colors

This module defines a type and tools for dealing with color information.

Each particle can have zero or more (in practice, usually not more than two) color indices. Color indices are positive; flow direction can be determined from the particle nature.

While parton shower matrix elements are diagonal in color, some special applications (e.g., subtractions for NLO matrix elements) require non-diagonal color matrices.

⟨colors.f90⟩≡
  ⟨*File header*⟩

  ```
  module colors
  ```

  ⟨*Use kinds*⟩
  ⟨*Use file utils*⟩
  ```
    use diagnostics !NODEP!
  ```

  ⟨*Standard module head*⟩

  ⟨*Colors: public*⟩

  ⟨*Colors: types*⟩

  ⟨*Colors: interfaces*⟩

  ```
  contains
  ```

  ⟨*Colors: procedures*⟩

  ```
  end module colors
  ```

7.2.1 The color type

A particle may have an arbitrary number of color indices (in practice, from zero to two, but more are possible). This object acts as a container.

The fact that color comes as an array prohibits elemental procedures in some places. (May add interfaces and multi versions where necessary.)

The color may be undefined; this corresponds to unallocated arrays.

NOTE: Due to a compiler bug in nagfor 5.2, we do not use allocatable but fixed-size arrays with dimension 2. Only nonzero entries count. This may be more efficient anyway, but gives up some flexibility. However, the squaring algorithm currently works only for singlets, (anti)triplets and octets anyway, so two components are enough.

⟨*Colors: public*⟩≡
  ```
    public :: color_t
  ```

⟨*Colors: types*⟩≡
  ```
    type :: color_t
       private
  !    integer, dimension(:), allocatable :: c1, c2
       integer, dimension(2) :: c1 = 0, c2 = 0
  ```

```
        logical :: ghost = .false.
    end type color_t
```

Initializers:

⟨*Colors: public*⟩+≡
```
    public :: color_init
```

⟨*Colors: interfaces*⟩≡
```
    interface color_init
       module procedure color_init_undefined, color_init_undefined_ghost
       module procedure color_init_array, color_init_array_ghost
       module procedure color_init_arrays, color_init_arrays_ghost
    end interface
```

Undefined color: array remains unallocated

⟨*Colors: procedures*⟩≡
```
    pure subroutine color_init_undefined (col)
      type(color_t), intent(out) :: col
    end subroutine color_init_undefined

    pure subroutine color_init_undefined_ghost (col, ghost)
      type(color_t), intent(out) :: col
      logical, intent(in) :: ghost
      col%ghost = ghost
    end subroutine color_init_undefined_ghost
```

This defines color from an arbitrary length color array, suitable for any representation. We may have two color arrays (non-diagonal matrix elements). This cannot be elemental. The third version assigns an array of colors, using a two-dimensional array as input.

⟨*Colors: procedures*⟩+≡
```
    pure subroutine color_init_array (col, c1)
      type(color_t), intent(out) :: col
      integer, dimension(:), intent(in) :: c1
 !    allocate (col%c1 (size (c1)))
 !    allocate (col%c2 (size (c1)))
      col%c1 = pack (c1, c1 /= 0, col%c1)
      col%c2 = col%c1
    end subroutine color_init_array

    pure subroutine color_init_array_ghost (col, c1, ghost)
      type(color_t), intent(out) :: col
      integer, dimension(:), intent(in) :: c1
      logical, intent(in) :: ghost
      call color_init_array (col, c1)
      col%ghost = ghost
    end subroutine color_init_array_ghost

    pure subroutine color_init_arrays (col, c1, c2)
      type(color_t), intent(out) :: col
      integer, dimension(:), intent(in) :: c1, c2
      if (size (c1) == size (c2)) then
```

```
!         allocate (col%c1 (size (c1)))
!         allocate (col%c2 (size (c2)))
          col%c1 = pack (c1, c1 /= 0, col%c1)
          col%c2 = pack (c2, c2 /= 0, col%c2)
       else if (size (c1) /= 0) then
!         allocate (col%c1 (size (c1)))
!         allocate (col%c2 (size (c1)))
          col%c1 = pack (c1, c1 /= 0, col%c1)
          col%c2 = col%c1
       else if (size (c2) /= 0) then
!         allocate (col%c1 (size (c2)))
!         allocate (col%c2 (size (c2)))
          col%c1 = pack (c2, c2 /= 0, col%c2)
          col%c2 = col%c1
       end if
     end subroutine color_init_arrays

     pure subroutine color_init_arrays_ghost (col, c1, c2, ghost)
       type(color_t), intent(out) :: col
       integer, dimension(:), intent(in) :: c1, c2
       logical, intent(in) :: ghost
       call color_init_arrays (col, c1, c2)
       col%ghost = ghost
     end subroutine color_init_arrays_ghost
```

This version is restricted to singlets, triplets, antitriplets, and octets: The input contains the color and anticolor index, each of the may be zero.

⟨*Colors: public*⟩+≡
```
     public :: color_init_col_acl
```

⟨*Colors: procedures*⟩+≡
```
     elemental subroutine color_init_col_acl (col, col_in, acl_in)
       type(color_t), intent(out) :: col
       integer, intent(in) :: col_in, acl_in
       integer, dimension(0) :: null_array
       select case (col_in)
       case (0)
          select case (acl_in)
          case (0)
             call color_init_array (col, null_array)
          case default
             call color_init_array (col, (/ -acl_in /))
          end select
       case default
          select case (acl_in)
          case (0)
             call color_init_array (col, (/ col_in /))
          case default
             call color_init_array (col, (/ col_in, -acl_in /))
          end select
       end select
     end subroutine color_init_col_acl
```

This version is used for the external interface. We convert a fixed-size array

of colors (for each particle) to the internal form by packing only the nonzero entries.

⟨*Colors: public*⟩+≡
```
    public :: color_init_from_array
```

⟨*Colors: interfaces*⟩+≡
```
    interface color_init_from_array
       module procedure color_init_from_array1, color_init_from_array1g
       module procedure color_init_from_array2, color_init_from_array2g
    end interface
```

⟨*Colors: procedures*⟩+≡
```
    pure subroutine color_init_from_array1 (col, c1)
      type(color_t), intent(out) :: col
      integer, dimension(:), intent(in) :: c1
      logical, dimension(size(c1)) :: mask
      mask = c1 /= 0
  !    allocate (col%c1 (count (mask)))
  !    allocate (col%c2 (size (col%c1)))
      col%c1 = pack (c1, mask, col%c1)
      col%c2 = col%c1
    end subroutine color_init_from_array1

    pure subroutine color_init_from_array1g (col, c1, ghost)
      type(color_t), intent(out) :: col
      integer, dimension(:), intent(in) :: c1
      logical, intent(in) :: ghost
      call color_init_from_array1 (col, c1)
      col%ghost = ghost
    end subroutine color_init_from_array1g

    pure subroutine color_init_from_array2 (col, c1)
      integer, dimension(:,:), intent(in) :: c1
      type(color_t), dimension(size(c1,2)), intent(out) :: col
      integer :: i
      do i = 1, size (c1,2)
         call color_init_from_array1 (col(i), c1(:,i))
      end do
    end subroutine color_init_from_array2

    pure subroutine color_init_from_array2g (col, c1, ghost)
      integer, dimension(:,:), intent(in) :: c1
      type(color_t), dimension(size(c1,2)), intent(out) :: col
      logical, intent(in), dimension(:) :: ghost
      call color_init_from_array2 (col, c1)
      col%ghost = ghost
    end subroutine color_init_from_array2g
```

Set the ghost property

⟨*Colors: public*⟩+≡
```
    public :: color_set_ghost
```

⟨*Colors: procedures*⟩+≡
```
    elemental subroutine color_set_ghost (col, ghost)
```

```
      type(color_t), intent(inout) :: col
      logical, intent(in) :: ghost
      col%ghost = ghost
    end subroutine color_set_ghost
```

Undefine the color state:

⟨*Colors: public*⟩+≡
```
    public :: color_undefine
```

⟨*Colors: procedures*⟩+≡
```
    elemental subroutine color_undefine (col, undefine_ghost)
      type(color_t), intent(inout) :: col
      logical, intent(in), optional :: undefine_ghost
  !   if (allocated (col%c1))  deallocate (col%c1)
  !   if (allocated (col%c2))  deallocate (col%c2)
      col%c1 = 0
      col%c2 = 0
      if (present (undefine_ghost)) then
         if (undefine_ghost)  col%ghost = .false.
      else
         col%ghost = .false.
      end if
    end subroutine color_undefine
```

Output. As dense as possible, no linebreak. If color is undefined, no output.

⟨*Colors: public*⟩+≡
```
    public :: color_write
```

⟨*Colors: interfaces*⟩+≡
```
    interface color_write
       module procedure color_write_single
       module procedure color_write_array
    end interface
```

⟨*Colors: procedures*⟩+≡
```
    subroutine color_write_single (col, unit)
      type(color_t), intent(in) :: col
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      if (color_is_defined (col)) then
         write (u, "(A)", advance="no")  "c("
         if (col%c1(1) /= 0)  write (u, "(I0)", advance="no")  col%c1(1)
         if (any (col%c1 /= 0))  write (u, "(1x)", advance="no")
         if (col%c1(2) /= 0)  write (u, "(I0)", advance="no")  col%c1(2)
         if (.not. color_is_diagonal (col)) then
            write (u, "(A)", advance="no")  "|"
            if (col%c2(1) /= 0)  write (u, "(I0)", advance="no")  col%c2(1)
            if (any (col%c2 /= 0))  write (u, "(1x)", advance="no")
            if (col%c2(2) /= 0)  write (u, "(I0)", advance="no")  col%c2(2)
         end if
         write (u, "(A)", advance="no")  ")"
      else if (col%ghost) then
         write (u, "(A)", advance="no")  "c*"
```

```
      end if
    end subroutine color_write_single

    subroutine color_write_array (col, unit)
      type(color_t), dimension(:), intent(in) :: col
      integer, intent(in), optional :: unit
      integer :: u
      integer :: i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)", advance="no") "["
      do i = 1, size (col)
         if (i > 1)  write (u, "(1x)", advance="no")
         call color_write_single (col(i), u)
      end do
      write (u, "(A)", advance="no") "]"
    end subroutine color_write_array
```

Binary I/O. For allocatable colors, this would have to be modified.

⟨*Colors: public*⟩+≡
```
    public :: color_write_raw
    public :: color_read_raw
```

⟨*Colors: procedures*⟩+≡
```
    subroutine color_write_raw (col, u)
      type(color_t), intent(in) :: col
      integer, intent(in) :: u
      logical :: defined
      defined = color_is_defined (col) .or. color_is_ghost (col)
      write (u) defined
      if (defined) then
         write (u) col%c1, col%c2
         write (u) col%ghost
      end if
    end subroutine color_write_raw

    subroutine color_read_raw (col, u, iostat)
      type(color_t), intent(out) :: col
      integer, intent(in) :: u
      integer, intent(out), optional :: iostat
      logical :: defined
      read (u, iostat=iostat) defined
      if (defined) then
         read (u, iostat=iostat) col%c1, col%c2
         read (u, iostat=iostat) col%ghost
      end if
    end subroutine color_read_raw
```

7.2.2   Predicates

Return the definition status

⟨*Colors: public*⟩+≡
```
    public :: color_is_defined
```

⟨*Colors: procedures*⟩+≡
```
    elemental function color_is_defined (col) result (defined)
      logical :: defined
      type(color_t), intent(in) :: col
  !    defined = allocated (col%c1)
      defined = any (col%c1 /= 0)
    end function color_is_defined
```

Diagonal color objects have only one array allocated:

⟨*Colors: public*⟩+≡
```
    public :: color_is_diagonal
```

⟨*Colors: procedures*⟩+≡
```
    elemental function color_is_diagonal (col) result (diagonal)
      logical :: diagonal
      type(color_t), intent(in) :: col
      if (color_is_defined (col)) then
         diagonal = all (col%c1 == col%c2)
      else
         diagonal = .true.
      end if
    end function color_is_diagonal
```

Return the ghost flag

⟨*Colors: public*⟩+≡
```
    public :: color_is_ghost
```

⟨*Colors: procedures*⟩+≡
```
    elemental function color_is_ghost (col) result (ghost)
      logical :: ghost
      type(color_t), intent(in) :: col
      ghost = col%ghost
    end function color_is_ghost
```

The ghost parity: true if the color-ghost flag is set.

⟨*Colors: interfaces*⟩+≡
```
    interface color_ghost_parity
       module procedure color_ghost_parity0
       module procedure color_ghost_parity1
    end interface
```

⟨*Colors: procedures*⟩+≡
```
    pure function color_ghost_parity0 (col) result (parity)
      type(color_t), intent(in) :: col
      logical :: parity
      parity = color_is_ghost (col)
    end function color_ghost_parity0

    pure function color_ghost_parity1 (col) result (parity)
      type(color_t), dimension(:), intent(in) :: col
      logical :: parity
      logical, dimension(size(col)) :: p
      integer :: i
      forall (i = 1:size(col))
```

```
      p(i) = color_ghost_parity0 (col(i))
    end forall
    parity = mod (count (p), 2) == 1
  end function color_ghost_parity1
```

7.2.3 Accessing contents

Return the number of color indices. We assume that it is identical for both arrays.

⟨*Colors: procedures*⟩+≡
```
    elemental function color_number (col) result (n)
      integer :: n
      type(color_t), intent(in) :: col
  !   n = size (col%c1)
      n = count (col%c1 /= 0)
    end function color_number
```

Return the (first) color/anticolor entry (assuming that color is diagonal). The result is a positive color index.

⟨*Colors: public*⟩+≡
```
    public :: color_get_col
    public :: color_get_acl
```

⟨*Colors: procedures*⟩+≡
```
    function color_get_col (col) result (c)
      integer :: c
      type(color_t), intent(in) :: col
      integer :: i
      do i = 1, size (col%c1)
        if (col%c1(i) > 0) then
          c = col%c1(i)
          return
        end if
      end do
      c = 0
    end function color_get_col

    function color_get_acl (col) result (c)
      integer :: c
      type(color_t), intent(in) :: col
      integer :: i
      do i = 1, size (col%c1)
        if (col%c1(i) < 0) then
          c = - col%c1(i)
          return
        end if
      end do
      c = 0
    end function color_get_acl
```

Return the color index with highest absolute value

⟨Colors: public⟩+≡
```
    public :: color_get_max_value
    !!! ifort 11.1 rev5 bug
    public :: color_get_max_value0
    public :: color_get_max_value1
    public :: color_get_max_value2
```

⟨Colors: interfaces⟩+≡
```
    interface color_get_max_value
       module procedure color_get_max_value0
       module procedure color_get_max_value1
       module procedure color_get_max_value2
    end interface
```

⟨Colors: procedures⟩+≡
```
    elemental function color_get_max_value0 (col) result (cmax)
      integer :: cmax
      type(color_t), intent(in) :: col
      cmax = maxval (abs (col%c1))
    end function color_get_max_value0

    pure function color_get_max_value1 (col) result (cmax)
      integer :: cmax
      type(color_t), dimension(:), intent(in) :: col
      cmax = maxval (color_get_max_value0 (col))
    end function color_get_max_value1

    function color_get_max_value2 (col) result (cmax)
      integer :: cmax
      type(color_t), dimension(:,:), intent(in) :: col
      integer, dimension(size(col, 2)) :: cm
      integer :: i
      forall (i = 1:size(col, 2))
         cm(i) = color_get_max_value1 (col(:,i))
      end forall
      cmax = maxval (cm)
    end function color_get_max_value2
```

7.2.4 Comparisons

Similar to helicities, colors match if they are equal, or if either one is undefined.

⟨Colors: public⟩+≡
```
    public :: operator(.match.)
    public :: operator(==)
    public :: operator(/=)
```

⟨Colors: interfaces⟩+≡
```
    interface operator(.match.)
       module procedure color_match
    end interface
    interface operator(==)
       module procedure color_eq
```

```fortran
      end interface
      interface operator(/=)
         module procedure color_neq
      end interface
```

⟨*Colors: procedures*⟩+≡
```fortran
    elemental function color_match (col1, col2) result (eq)
      logical :: eq
      type(color_t), intent(in) :: col1, col2
      if (color_is_defined (col1) .and. color_is_defined (col2)) then
!        if (size (col1%c1) == size (col2%c1)) then
           eq = all (col1%c1 == col2%c1) .and. all (col1%c2 == col2%c2)
!        else
!           eq = .false.
!        end if
      else
         eq = .true.
      end if
    end function color_match

    elemental function color_eq (col1, col2) result (eq)
      logical :: eq
      type(color_t), intent(in) :: col1, col2
      if (color_is_defined (col1) .and. color_is_defined (col2)) then
!        if (size (col1%c1) == size (col2%c1)) then
           eq = all (col1%c1 == col2%c1) .and. all (col1%c2 == col2%c2) &
                .and. (col1%ghost .eqv. col2%ghost)
!        else
!           eq = .false.
!        end if
      else if (.not. color_is_defined (col1) &
         .and. .not. color_is_defined (col2)) then
         eq = col1%ghost .eqv. col2%ghost
      else
         eq = .false.
      end if
    end function color_eq
```

⟨*Colors: procedures*⟩+≡
```fortran
    elemental function color_neq (col1, col2) result (neq)
      logical :: neq
      type(color_t), intent(in) :: col1, col2
      if (color_is_defined (col1) .and. color_is_defined (col2)) then
!        if (size (col1%c1) == size (col2%c1)) then
           neq = any (col1%c1 /= col2%c1) .or. any (col1%c2 /= col2%c2) &
                .or. (col1%ghost .neqv. col2%ghost)
!        else
!           neq = .true.
!        end if
      else if (.not. color_is_defined (col1) &
         .and. .not. color_is_defined (col2)) then
         neq = col1%ghost .neqv. col2%ghost
      else
         neq = .true.
```

```
      end if
   end function color_neq
```

7.2.5 Tools

Shift color indices by a common offset.

⟨*Colors: public*⟩+≡
```
    public :: color_add_offset
```

⟨*Colors: procedures*⟩+≡
```
    elemental subroutine color_add_offset (col, offset)
      type(color_t), intent(inout) :: col
      integer, intent(in) :: offset
      where (col%c1 /= 0)  col%c1 = col%c1 + sign (offset, col%c1)
      where (col%c2 /= 0)  col%c2 = col%c2 + sign (offset, col%c2)
  !    if (allocated (col%c1)) then
  !       col%c1 = col%c1 + sign (offset, col%c1)
  !       col%c2 = col%c2 + sign (offset, col%c2)
  !    end if
    end subroutine color_add_offset
```

Reassign color indices for an array of colored particle in canonical order. The
allocated size of the color map is such that two colors per particle can be acco-
modated.

⟨*Colors: public*⟩+≡
```
    public :: color_canonicalize
```

⟨*Colors: procedures*⟩+≡
```
    subroutine color_canonicalize (col)
      type(color_t), dimension(:), intent(inout) :: col
      integer, dimension(2*size(col)) :: map
      integer :: n_col, i, j, k
      n_col = 0
      do i = 1, size (col)
         do j = 1, size (col(i)%c1)
            if (col(i)%c1(j) /= 0) then
               k = find (abs (col(i)%c1(j)), map(:n_col))
               if (k == 0) then
                  n_col = n_col + 1
                  map(n_col) = abs (col(i)%c1(j))
                  k = n_col
               end if
               col(i)%c1(j) = sign (k, col(i)%c1(j))
            end if
            if (col(i)%c2(j) /= 0) then
               k = find (abs (col(i)%c2(j)), map(:n_col))
               if (k == 0) then
                  n_col = n_col + 1
                  map(n_col) = abs (col(i)%c2(j))
                  k = n_col
               end if
               col(i)%c2(j) = sign (k, col(i)%c2(j))
```

```
          end if
        end do
      end do
  contains
    function find (c, array) result (k)
      integer :: k
      integer, intent(in) :: c
      integer, dimension(:), intent(in) :: array
      integer :: i
      k = 0
      do i = 1, size (array)
        if (c == array (i)) then
          k = i
          return
        end if
      end do
    end function find
  end subroutine color_canonicalize
```

Return an array of different color indices from an array of colors. The last argument is a pseudo-color array, where the color entries correspond to the position of the corresponding index entry in the index array. The colors are assumed to be diagonal.

⟨*Colors: procedures*⟩+≡

```
  subroutine extract_color_line_indices (col, c_index, col_pos)
    type(color_t), dimension(:), intent(in) :: col
    integer, dimension(:), intent(out), allocatable :: c_index
    type(color_t), dimension(size(col)), intent(out) :: col_pos
    integer, dimension(:), allocatable :: c_tmp
    integer :: i, j, k, n, c
    allocate (c_tmp (sum (color_number (col))))
    n = 0
    SCAN1: do i = 1, size (col)
      SCAN2: do j = 1, 2
        c = abs (col(i)%c1(j))
        if (c /= 0) then
          do k = 1, n
            if (c_tmp(k) == c) then
              col_pos(i)%c1(j) = k
              cycle SCAN2
            end if
          end do
          n = n + 1
          c_tmp(n) = c
          col_pos(i)%c1(j) = n
        end if
      end do SCAN2
    end do SCAN1
    allocate (c_index (n))
    c_index = c_tmp(1:n)
  end subroutine extract_color_line_indices
```

Given a color array, pairwise contract the color lines in all possible ways and

return the resulting array of arrays. The input color array must be diagonal, and each color should occur exactly twice, once as color and once as anticolor.

Gluon entries with equal color and anticolor are explicitly excluded.

This algorithm is generic, but for long arrays it is neither efficient, nor does it avoid duplicates. It is intended for small arrays, in particular for the state matrix of a structure-function pair.

⟨*Colors: public*⟩+≡
```
    public :: color_array_make_contractions
```

⟨*Colors: procedures*⟩+≡
```
    subroutine color_array_make_contractions (col_in, col_out)
      type(color_t), dimension(:), intent(in) :: col_in
      type(color_t), dimension(:,:), intent(out), allocatable :: col_out
      type :: entry_t
         integer, dimension(:), allocatable :: map
         type(color_t), dimension(:), allocatable :: col
         type(entry_t), pointer :: next => null ()
      end type entry_t
      type :: list_t
         integer :: n = 0
         type(entry_t), pointer :: first => null ()
         type(entry_t), pointer :: last => null ()
      end type list_t
      type(list_t) :: list
      type(entry_t), pointer :: entry
      integer, dimension(:), allocatable :: c_index
      type(color_t), dimension(size(col_in)) :: col_pos
      integer :: n_prt, n_c_index
      integer, dimension(:), allocatable :: map
      integer :: i, j, c
      n_prt = size (col_in)
      call extract_color_line_indices (col_in, c_index, col_pos)
      ! print *, c_index
      n_c_index = size (c_index)
      allocate (map (n_c_index))
      map = 0
      call list_append_if_valid (list, map)
      entry => list%first
      do while (associated (entry))
         do i = 1, n_c_index
            if (entry%map(i) == 0) then
               c = c_index(i)
               do j = i + 1, n_c_index
                  if (entry%map(j) == 0) then
                     map = entry%map
                     map(i) = c
                     map(j) = c
                     call list_append_if_valid (list, map)
                  end if
               end do
            end if
         end do
         entry => entry%next
      end do
```

```fortran
      call list_to_array (list, col_out)
contains
  subroutine list_append_if_valid (list, map)
    type(list_t), intent(inout) :: list
    integer, dimension(:), intent(in) :: map
    type(entry_t), pointer :: entry
    integer :: i, j, c, p
    entry => list%first
    do while (associated (entry))
       if (all (map == entry%map))  return
       entry => entry%next
    end do
    allocate (entry)
    allocate (entry%map (n_c_index))
    entry%map = map
    allocate (entry%col (n_prt))
    do i = 1, n_prt
       do j = 1, 2
          c = col_in(i)%c1(j)
          if (c /= 0) then
             p = col_pos(i)%c1(j)
             if (map(p) /= 0) then
                entry%col(i)%c1(j) = sign (map(p), c)
             else
                entry%col(i)%c1(j) = c
             endif
             entry%col(i)%c2(j) = entry%col(i)%c1(j)
          end if
       end do
       if (any (entry%col(i)%c1 /= 0) .and. &
            entry%col(i)%c1(1) == - entry%col(i)%c1(2))  return
    end do
    ! call color_write (entry%col); print *, map
    if (associated (list%last)) then
       list%last%next => entry
    else
       list%first => entry
    end if
    list%last => entry
    list%n = list%n + 1
  end subroutine list_append_if_valid
  subroutine list_to_array (list, col)
    type(list_t), intent(inout) :: list
    type(color_t), dimension(:,:), intent(out), allocatable :: col
    type(entry_t), pointer :: entry
    integer :: i
    allocate (col (n_prt, list%n - 1))
    do i = 0, list%n - 1
       entry => list%first
       list%first => list%first%next
       if (i /= 0)  col(:,i) = entry%col
       deallocate (entry)
    end do
    list%last => null ()
```

```
      end subroutine list_to_array
    end subroutine color_array_make_contractions
```

Invert the color index, switching from particle to antiparticle. For gluons, we
have to swap the order of color entries.

⟨*Colors: public*⟩+≡
```
    public :: color_invert
```

⟨*Colors: procedures*⟩+≡
```
    elemental subroutine color_invert (col)
      type(color_t), intent(inout) :: col
      col%c1 = - col%c1
      col%c2 = - col%c2
      if (col%c1(1) < 0 .and. col%c1(2) > 0) then
         col%c1 = col%c1(2:1:-1)
         col%c2 = col%c2(2:1:-1)
      end if
    end subroutine color_invert
```

Make a color map for two matching color arrays. The result is an array of
integer pairs.

⟨*Colors: public*⟩+≡
```
    public :: set_color_map
```

⟨*Colors: procedures*⟩+≡
```
    subroutine set_color_map (map, col1, col2)
      integer, dimension(:,:), intent(out), allocatable :: map
      type(color_t), dimension(:), intent(in) :: col1, col2
      integer, dimension(:,:), allocatable :: map1
      integer :: i, j, k
      allocate (map1 (2, 2 * sum (color_number (col1))))
      k = 0
      do i = 1, size (col1)
         do j = 1, size (col1(i)%c1)
            if (col1(i)%c1(j) /= 0 &
                 .and. all (map1(1,:k) /= abs (col1(i)%c1(j)))) then
               k = k + 1
               map1(1,k) = abs (col1(i)%c1(j))
               map1(2,k) = abs (col2(i)%c1(j))
            end if
            if (col1(i)%c2(j) /= 0 &
                 .and. all (map1(1,:k) /= abs (col1(i)%c2(j)))) then
               k = k + 1
               map1(1,k) = abs (col1(i)%c2(j))
               map1(2,k) = abs (col2(i)%c2(j))
            end if
         end do
      end do
      allocate (map (2, k))
      map(:,:) = map1(:,:k)
    end subroutine set_color_map
```

Translate colors which have a match in the translation table (an array of integer pairs). Color that do not match an entry are simply transferred; this is done by first transferring all components, then modifying entries where appropriate.

⟨*Colors: public*⟩+≡
```
    public :: color_translate
```

⟨*Colors: interfaces*⟩+≡
```
    interface color_translate
       module procedure color_translate0
       module procedure color_translate0_offset
       module procedure color_translate1
    end interface
```

⟨*Colors: procedures*⟩+≡
```
    subroutine color_translate0 (col, map)
      type(color_t), intent(inout) :: col
      integer, dimension(:,:), intent(in) :: map
      type(color_t) :: col_tmp
      integer :: i
      col_tmp = col
      do i = 1, size (map,2)
         where (abs (col%c1) == map(1,i))
            col_tmp%c1 = sign (map(2,i), col%c1)
         end where
         where (abs (col%c2) == map(1,i))
            col_tmp%c2 = sign (map(2,i), col%c2)
         end where
      end do
      col = col_tmp
    end subroutine color_translate0

    subroutine color_translate0_offset (col, map, offset)
      type(color_t), intent(inout) :: col
      integer, dimension(:,:), intent(in) :: map
      integer, intent(in) :: offset
      logical, dimension(size(col%c1)) :: mask1, mask2
      type(color_t) :: col_tmp
      integer :: i
      col_tmp = col
      mask1 = col%c1 /= 0
      mask2 = col%c2 /= 0
      do i = 1, size (map,2)
         where (abs (col%c1) == map(1,i))
            col_tmp%c1 = sign (map(2,i), col%c1)
            mask1 = .false.
         end where
         where (abs (col%c2) == map(1,i))
            col_tmp%c2 = sign (map(2,i), col%c2)
            mask2 = .false.
         end where
      end do
      col = col_tmp
      where (mask1)  col%c1 = sign (abs (col%c1) + offset, col%c1)
      where (mask2)  col%c2 = sign (abs (col%c2) + offset, col%c2)
```

```
      end subroutine color_translate0_offset

    subroutine color_translate1 (col, map, offset)
      type(color_t), dimension(:), intent(inout) :: col
      integer, dimension(:,:), intent(in) :: map
      integer, intent(in), optional :: offset
      integer :: i
      if (present (offset)) then
         do i = 1, size (col)
            call color_translate0_offset (col(i), map, offset)
         end do
      else
         do i = 1, size (col)
            call color_translate0 (col(i), map)
         end do
      end if
    end subroutine color_translate1
```

Merge two color objects by taking the first entry from the first and the second
entry from the second argument. Makes sense only if the input colors are defined
(and diagonal). If either one is undefined, transfer the defined one.

For a color ghost, color is not defined. These have to be treated separately.

⟨*Colors: public*⟩+≡
```
    public :: operator(.merge.)
```

⟨*Colors: interfaces*⟩+≡
```
    interface operator(.merge.)
       module procedure merge_colors
    end interface
```

⟨*Colors: procedures*⟩+≡
```
    elemental function merge_colors (col1, col2) result (col)
      type(color_t) :: col
      type(color_t), intent(in) :: col1, col2
      if (color_is_defined (col1) .and. color_is_defined (col2)) then
         call color_init_arrays (col, col1%c1, col2%c1)
      else if (color_is_defined (col1)) then
         col = col1
      else if (color_is_defined (col2)) then
         col = col2
      else if (color_is_ghost (col1)) then
         col = col1
      else if (color_is_ghost (col2)) then
         col = col2
      end if
    end function merge_colors
```

Compute the color factor, given two interfering color arrays.

⟨*Colors: public*⟩+≡
```
    public :: compute_color_factor
```

⟨*Colors: procedures*⟩+≡
```
    function compute_color_factor (col1, col2, nc) result (factor)
      real(default) :: factor
```

```
      type(color_t), dimension(:), intent(in) :: col1, col2
      integer, intent(in), optional :: nc
      type(color_t), dimension(size(col1)) :: col
      integer :: ncol, nloops, nghost
      ncol = 3;  if (present (nc))  ncol = nc
      col = col1 .merge. col2
      nloops = count_color_loops (col)
      nghost = count (color_is_ghost (col))
      factor = real (ncol, default) ** (nloops - nghost)
      if (color_ghost_parity (col))  factor = - factor
    end function compute_color_factor
```

We have a pair of color index arrays which corresponds to a squared matrix element. We want to determine the number of color loops in this square matrix element. So we first copy the colors (stored in a single color array with a pair of color lists in each entry) to a temporary where the color indices are shifted by some offset. We then recursively follow each loop, starting at the first color that has the offset, resetting the first color index to the loop index and each further index to zero as we go. We check that (a) each color index occurs twice within the left (right) color array, (b) the loops are closed, so we always come back to a line which has the loop index.

In order for the algorithm to work we have to conjugate the colors of initial state particles (one for decays, two for scatterings) into their corresponding anticolors of outgoing particles.

⟨*Colors: public*⟩+≡
```
    public :: count_color_loops
```

⟨*Colors: procedures*⟩+≡
```
    function count_color_loops (col) result (count)
      integer :: count
      type(color_t), dimension(:), intent(in) :: col
      type(color_t), dimension(size(col)) :: cc
      integer :: i, n, offset
!     print *, "Count color loops:"
!     call color_write (col); print *
      cc = col
      n = size (cc)
      offset = n
      call color_add_offset (cc, offset)
!     print *, offset
!     call color_write (cc); print *
      count = 0
      SCAN_LOOPS: do
        do i = 1, n
!         print *, i, ':', cc(i)%c1
          if (color_is_defined (cc(i))) then
            if (any (cc(i)%c1 > offset)) then
!             print *, 'start', i
              count = count + 1
              call follow_line1 (pick_new_line (cc(i)%c1, count, 1))
              cycle SCAN_LOOPS
            end if
          end if
```

```fortran
      end do
      exit SCAN_LOOPS
   end do SCAN_LOOPS
contains
   function pick_new_line (c, reset_val, sgn) result (line)
      integer :: line
      integer, dimension(:), intent(inout) :: c
      integer, intent(in) :: reset_val
      integer, intent(in) :: sgn
      integer :: i
      if (any (c == count)) then
         line = count
      else
         do i = 1, size (c)
            if (sign (1, c(i)) == sgn .and. abs (c(i)) > offset) then
               line = c(i)
               c(i) = reset_val
               return
            end if
         end do
         call color_mismatch
      end if
   end function pick_new_line
   subroutine reset_line (c, line)
      integer, dimension(:), intent(inout) :: c
      integer, intent(in) :: line
      integer :: i
      do i = 1, size (c)
         if (c(i) == line) then
            c(i) = 0
            return
         end if
      end do
   end subroutine reset_line
   recursive subroutine follow_line1 (line)
      integer, intent(in) :: line
      integer :: i
!     print *, 'follow line 1:', line
      if (line == count) then
!        print *, 'loop closed'
         return
      end if
      do i = 1, n
         if (any (cc(i)%c1 == -line)) then
            call reset_line (cc(i)%c1, -line)
!           print *, 'found', -line, ' resetting c1:'
!           call color_write (cc); print *
            call follow_line2 (pick_new_line (cc(i)%c2, 0, sign (1, -line)))
            return
         end if
      end do
      call color_mismatch ()
   end subroutine follow_line1
   recursive subroutine follow_line2 (line)
```

524

```
        integer, intent(in) :: line
        integer :: i
!        print *, 'follow line 2:', line
        do i = 1, n
           if (any (cc(i)%c2 == -line)) then
              call reset_line (cc(i)%c2, -line)
!             print *, 'found', -line, ' resetting c2:'
!             call color_write (cc); print *
              call follow_line1 (pick_new_line (cc(i)%c1, 0, sign (1, -line)))
              return
           end if
        end do
        call color_mismatch ()
      end subroutine follow_line2
      subroutine color_mismatch ()
        call color_write (col)
        print *
        call msg_bug (" Color flow mismatch (color loops should be closed)")
      end subroutine color_mismatch
    end function count_color_loops
```

7.2.6 Color counting test

⟨*Colors: public*⟩+≡
```
    public :: color_test
```

⟨*Colors: procedures*⟩+≡
```
    subroutine color_test ()
      type(color_t), dimension(4) :: col1, col2, col
      type(color_t), dimension(:), allocatable :: col3
      type(color_t), dimension(:,:), allocatable :: col_array
      integer :: count, i
      call color_init_col_acl (col1, (/ 1, 0, 2, 3 /), (/ 0, 1, 3, 2 /))
      col2 = col1
      call color_write (col1); print *
      call color_write (col2); print *
      col = col1 .merge. col2
      call color_write (col); print *
      count = count_color_loops (col)
      print *, "Number of color loops (3): ", count
      call color_init_col_acl (col2, (/ 1, 0, 2, 3 /), (/ 0, 2, 3, 1 /))
      call color_write (col1); print *
      call color_write (col2); print *
      col = col1 .merge. col2
      call color_write (col); print *
      count = count_color_loops (col)
      print *, "Number of color loops (2): ", count
      print *
      allocate (col3 (4))
      call color_init_from_array (col3, &
           reshape ((/ 1, 0,   0, -1,  2, -3,  3, -2 /), &
                    (/ 2, 4 /)))
      call color_write (col3); print *
```

```
      call color_array_make_contractions (col3, col_array)
      print *, "Contractions:"
      do i = 1, size (col_array, 2)
         call color_write (col_array(:,i)); print *
      end do
      deallocate (col3)
      print *
      allocate (col3 (6))
      call color_init_from_array (col3, &
           reshape ((/ 1, -2,   3, 0,   0, -1,   2, -4,   -3, 0,   4, 0 /), &
                    (/ 2, 6 /)))
      call color_write (col3); print *
      call color_array_make_contractions (col3, col_array)
      print *, "Contractions:"
      do i = 1, size (col_array, 2)
         call color_write (col_array(:,i)); print *
      end do
   end subroutine color_test
```

7.2.7 The Madgraph color model

This section describes the method for matrix element and color flow calculation within Madgraph.

For each Feynman diagram, the colorless amplitude for a specified helicity and momentum configuration (in- and out- combined) is computed:

$$A_d(p, h) \tag{7.1}$$

Inserting color, the squared matrix element for definite helicity and momentum is

$$M^2(p, h) = \sum_{dd'} A_d(p, h) \, C_{dd'} A_{d'}^*(p, h) \tag{7.2}$$

where $C_{dd'}$ describes the color interference of the two diagrams A_d and A_d', which is independent of momentum and helicity and can be calculated for each Feynman diagram pair by reducing it to the corresponding color graph. Obviously, one could combine all diagrams with identical color structure, such that the index d runs only over different color graphs. For colorless diagrams all elements of $C_{dd'}$ are equal to unity.

The hermitian matrix $C_{dd'}$ is diagonalized once and for all, such that it can be written in the form

$$C_{dd'} = \sum_{\lambda} c_d^\lambda \lambda \, c_d^{\lambda *}, \tag{7.3}$$

where the eigenvectors c_d are normalized,

$$\sum_d |c_d^\lambda|^2 = 1, \tag{7.4}$$

and the λ values are the corresponding eigenvalues. In the colorless case, this means $c_d = 1/\sqrt{N_d}$ for all diagrams (N_d = number of diagrams), and $\lambda = N_d$ is the only nonzero eigenvalue.

Consequently, the squared matrix element for definite helicity and momentum can also be written as

$$M^2(p,h) = \sum_\lambda A_\lambda(p,h) \, \lambda \, A_\lambda(p,h)^* \tag{7.5}$$

with

$$A_\lambda(p,h) = \sum_d c_d^\lambda A_d(p,h). \tag{7.6}$$

For generic spin density matrices, this is easily generalized to

$$M^2(p,h,h') = \sum_\lambda A_\lambda(p,h) \, \lambda \, A_\lambda(p,h')^* \tag{7.7}$$

To determine the color flow probabilities of a given momentum-helicity configuration, the color flow amplitudes are calculated as

$$a_f(p,h) = \sum_d \beta_d^f A_d(p,h), \tag{7.8}$$

where the coefficients β_d^f describe the amplitude for a given Feynman diagram (or color graph) d to correspond to a definite color flow f. They are computed from $C_{dd'}$ by transforming this matrix into the color flow basis and neglecting all off-diagonal elements. Again, these coefficients do not depend on momentum or helicity and can therefore be calculated in advance. This gives the color flow transition matrix

$$F^f(p,h,h') = a_f(p,h) \, a_f^*(p,h') \tag{7.9}$$

which is assumed diagonal in color flow space and is separate from the color-summed transition matrix M^2. They are, however, equivalent (up to a factor) to leading order in $1/N_c$, and using the color flow transition matrix is appropriate for matching to hadronization.

Note that the color flow transition matrix is not normalized at this stage. To make use of it, we have to fold it with the in-state density matrix to get a pseudo density matrix

$$\hat\rho_{\text{out}}^f(p,h_{\text{out}},h'_{\text{out}}) = \sum_{h_{\text{in}}h'_{\text{in}}} F^f(p,h,h') \, \rho_{\text{in}}(p,h_{\text{in}},h'_{\text{in}}) \tag{7.10}$$

which gets a meaning only after contracted with projections on the outgoing helicity states k_{out}, given as linear combinations of helicity states with the unitary coefficient matrix $c(k_{\text{out}}, h_{\text{out}})$. Then the probability of finding color flow f when the helicity state k_{out} is measured is given by

$$P^f(p,k_{\text{out}}) = Q^f(p,k_{\text{out}}) / \sum_f Q^f(p,k_{\text{out}}) \tag{7.11}$$

where

$$Q^f(p,k_{\text{out}}) = \sum_{h_{\text{out}}h'_{\text{out}}} c(k_{\text{out}},h_{\text{out}}) \, \hat\rho_{\text{out}}^f(p,h_{\text{out}},h'_{\text{out}}) \, c^*(k_{\text{out}},h'_{\text{out}}) \tag{7.12}$$

However, if we can assume that the out-state helicity basis is the canonical one, we can throw away the off diagonal elements in the color flow density matrix and normalize the ones on the diagonal to obtain

$$P^f(p, h_{\text{out}}) = \hat{\rho}^f_{\text{out}}(p, h_{\text{out}}, h_{\text{out}}) / \sum_f \hat{\rho}^f_{\text{out}}(p, h_{\text{out}}, h_{\text{out}}) \tag{7.13}$$

Finally, the color-summed out-state density matrix is computed by the scattering formula

$$\rho_{\text{out}}(p, h_{\text{out}}, h'_{\text{out}}) = \sum_{h_{\text{in}} h'_{\text{in}}} M^2(p, h, h') \, \rho_{\text{in}}(p, h_{\text{in}}, h'_{\text{in}}) \tag{7.14}$$

$$= \sum_{h_{\text{in}} h'_{\text{in}} \lambda} A_\lambda(p, h) \, \lambda \, A_\lambda(p, h')^* \rho_{\text{in}}(p, h_{\text{in}}, h'_{\text{in}}), \tag{7.15}$$

The trace of ρ_{out} is the squared matrix element, summed over all internal degrees of freedom. To get the squared matrix element for a definite helicity k_{out} and color flow f, one has to project the density matrix onto the given helicity state and multiply with $P^f(p, k_{\text{out}})$.

For diagonal helicities the out-state density reduces to

$$\rho_{\text{out}}(p, h_{\text{out}}) = \sum_{h_{\text{in}} \lambda} \lambda |A_\lambda(p, h)|^2 \rho_{\text{in}}(p, h_{\text{in}}). \tag{7.16}$$

Since no basis transformation is involved, we can use the normalized color flow probability $P^f(p, h_{\text{out}})$ and express the result as

$$\rho^f_{\text{out}}(p, h_{\text{out}}) = \rho_{\text{out}}(p, h_{\text{out}}) \, P^f(p, h_{\text{out}}) \tag{7.17}$$

$$= \sum_{h_{\text{in}} \lambda} \frac{|a^f(p, h)|^2}{\sum_f |a^f(p, h)|^2} \lambda |A_\lambda(p, h)|^2 \rho_{\text{in}}(p, h_{\text{in}}). \tag{7.18}$$

From these considerations, the following calculation strategy can be derived:

- Before the first event is generated, the color interference matrix $C_{dd'}$ is computed and diagonalized, so the eigenvectors c^λ_d, eigenvalues λ and color flow coefficients β^f_d are obtained. In practice, these calculations are done when the matrix element code is generated, and the results are hardcoded in the matrix element subroutine as `DATA` statements.

- For each event, one loops over helicities once and stores the matrices $A_\lambda(p, h)$ and $a^f(p, h)$. The allowed color flows, helicity combinations and eigenvalues are each labeled by integer indices, so one has to store complex matrices of dimension $N_\lambda \times N_h$ and $N_f \times N_h$, respectively.

- The further strategy depends on the requested information.

  1. If colorless diagonal helicity amplitudes are required, the eigenvalues $A_\lambda(p, h)$ are squared, summed with weight λ, and the result contracted with the in-state probability vector $\rho_{\text{in}}(p, h_{\text{in}})$. The result is a probability vector $\rho_{\text{out}}(p, h_{\text{out}})$.

2. For colored diagonal helicity amplitudes, the color coefficients $a^f(p, h)$ are also squared and used as weights to obtain the color-flow probability vector $\rho_{\text{out}}^f(p, h_{\text{out}})$.

3. For colorless non-diagonal helicity amplitudes, we contract the tensor product of $A_\lambda(p, h)$ with $A_\lambda(p, h')$, weighted with λ, with the correlated in-state density matrix, to obtain a correlated out-state density matrix.

4. In the general (colored, non-diagonal) case, we do the same as in the colorless case, but return the un-normalized color flow density matrix $\hat{\rho}_{\text{out}}^f(p, h_{\text{out}}, h'_{\text{out}})$ in addition. When the relevant helicity basis is known, the latter can be used by the caller program to determine flow probabilities. (In reality, we assume the canonical basis and reduce the correlated out-state density to its diagonal immediately.)

7.3 Flavors: Particle properties

This module contains a type for holding the flavor code, and all functions that depend on the model, i.e., that determine particle properties.

The PDG code is packed in a special `flavor` type. (This prohibits meaningless operations, and it allows for a different implementation, e.g., some non-PDG scheme internally, if appropiate at some point.) In addition, the flavor object holds a pointer to a `particle_data` object which is centrally stored (as part of a physics model). In this way, all particle data can be accessed using just the `flv` object without having to carry them around.

The pointer component imposes a technical restriction: Assignment of flavor objects cannot be used, directly or indirectly, in pure, and thus in elemental procedures.

⟨`flavors.f90`⟩≡
  ⟨*File header*⟩

  module flavors

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use pdg_arrays
    use colors, only: color_t, color_init
    use models

  ⟨*Standard module head*⟩

  ⟨*Flavors: public*⟩

  ⟨*Flavors: types*⟩

  ⟨*Flavors: interfaces*⟩

  contains

  ⟨*Flavors: procedures*⟩

  end module flavors

7.3.1 The flavor type

The flavor type is an integer representing the PDG code, or undefined (zero). Negative codes represent antiflavors.

For full generality, and analogy with helicity and color, we allow for non-diagonal flavor indices in density matrices. This is probably academic; an obscure application is the definition of proper isospin states.

Further properties of the given flavor can be retrieved via the particle-data pointer, if it is associated.

⟨*Flavors: public*⟩≡
    public :: flavor_t

⟨*Flavors: types*⟩≡

```
type :: flavor_t
   private
   integer :: f = UNDEFINED
   type(particle_data_t), pointer :: prt => null ()
end type flavor_t
```

Initializer form. If the model is assigned, the procedure is impure, therefore we have to define a separate array version.

⟨*Flavors: public*⟩+≡

```
public :: flavor_init
```

⟨*Flavors: interfaces*⟩≡

```
interface flavor_init
   module procedure flavor_init0_empty
   module procedure flavor_init0
   module procedure flavor_init0_particle_data
   module procedure flavor_init0_model
   module procedure flavor_init0_name_model
   module procedure flavor_init1_model
   module procedure flavor_init1_name_model
   module procedure flavor_init2_model
   module procedure flavor_init_aval_model
end interface
```

⟨*Flavors: procedures*⟩≡

```
elemental subroutine flavor_init0_empty (flv)
  type(flavor_t), intent(out) :: flv
end subroutine flavor_init0_empty

elemental subroutine flavor_init0 (flv, f)
  type(flavor_t), intent(out) :: flv
  integer, intent(in) :: f
  flv%f = f
end subroutine flavor_init0

subroutine flavor_init0_particle_data (flv, particle_data)
  type(flavor_t), intent(out) :: flv
  type(particle_data_t), intent(in), target :: particle_data
  flv%f = particle_data_get_pdg (particle_data)
  flv%prt => particle_data
end subroutine flavor_init0_particle_data

subroutine flavor_init0_model (flv, f, model)
  type(flavor_t), intent(out) :: flv
  integer, intent(in) :: f
  type(model_t), intent(in), target :: model
  flv%f = f
  flv%prt => model_get_particle_ptr (model, f)
end subroutine flavor_init0_model

subroutine flavor_init1_model (flv, f, model)
  type(flavor_t), dimension(:), intent(out) :: flv
  integer, dimension(:), intent(in) :: f
  type(model_t), intent(in), target :: model
```

```
        integer :: i
        do i = 1, size (f)
           call flavor_init0_model (flv(i), f(i), model)
        end do
    end subroutine flavor_init1_model

    subroutine flavor_init2_model (flv, f, model)
      type(flavor_t), dimension(:,:), intent(out) :: flv
      integer, dimension(:,:), intent(in) :: f
      type(model_t), intent(in), target :: model
      integer :: i
      do i = 1, size (f, 2)
         call flavor_init1_model (flv(:,i), f(:,i), model)
      end do
    end subroutine flavor_init2_model

    subroutine flavor_init0_name_model (flv, name, model)
      type(flavor_t), intent(out) :: flv
      type(string_t), intent(in) :: name
      type(model_t), intent(in), target :: model
      flv%f = model_get_particle_pdg (model, name)
      flv%prt => model_get_particle_ptr (model, flv%f)
    end subroutine flavor_init0_name_model

    subroutine flavor_init1_name_model (flv, name, model)
      type(flavor_t), dimension(:), intent(out) :: flv
      type(string_t), dimension(:), intent(in) :: name
      type(model_t), intent(in), target :: model
      integer :: i
      do i = 1, size (name)
         call flavor_init0_name_model (flv(i), name(i), model)
      end do
    end subroutine flavor_init1_name_model
```

This version transforms a PDG array value into a flavor array. The flavor array must be allocatable.

⟨*Flavors: procedures*⟩+≡

```
    subroutine flavor_init_aval_model (flv, aval, model)
      type(flavor_t), dimension(:), intent(out), allocatable :: flv
      type(pdg_array_t), intent(in) :: aval
      type(model_t), intent(in), target :: model
      integer, dimension(:), allocatable :: pdg
      pdg = aval
      allocate (flv (size (pdg)))
      call flavor_init (flv, pdg, model)
    end subroutine flavor_init_aval_model
```

Undefine the flavor state:

⟨*Flavors: public*⟩+≡

```
    public :: flavor_undefine
```

⟨*Flavors: procedures*⟩+≡

```
    elemental subroutine flavor_undefine (flv)
```

```
      type(flavor_t), intent(inout) :: flv
      flv%f = UNDEFINED
    end subroutine flavor_undefine
```

Output: dense, no linebreak

⟨*Flavors: public*⟩+≡
```
    public :: flavor_write
```

⟨*Flavors: procedures*⟩+≡
```
    subroutine flavor_write (flv, unit)
      type(flavor_t), intent(in) :: flv
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      if (associated (flv%prt)) then
         write (u, "(A)", advance="no")  "f("
      else
         write (u, "(A)", advance="no")  "p("
      end if
      write (u, "(I0)", advance="no")  flv%f
      write (u, "(A)", advance="no")  ")"
    end subroutine flavor_write
```

Binary I/O. Currently, the model information is not written/read, so after reading the particle-data pointer is empty.

⟨*Flavors: public*⟩+≡
```
    public :: flavor_write_raw
    public :: flavor_read_raw
```

⟨*Flavors: procedures*⟩+≡
```
    subroutine flavor_write_raw (flv, u)
      type(flavor_t), intent(in) :: flv
      integer, intent(in) :: u
      write (u) flv%f
    end subroutine flavor_write_raw

    subroutine flavor_read_raw (flv, u, iostat)
      type(flavor_t), intent(out) :: flv
      integer, intent(in) :: u
      integer, intent(out), optional :: iostat
      read (u, iostat=iostat) flv%f
    end subroutine flavor_read_raw
```

Assignment

Default assignment of flavor objects is possible, but cannot be used in pure procedures, because a pointer assignment is involved.

Assign the particle pointer separately. This cannot be elemental, so we define a scalar and an array version explicitly. We refer to an array of flavors, not an array of models.

⟨*Flavors: public*⟩+≡
```
    public :: flavor_set_model
```

⟨*Flavors: interfaces*⟩+≡

```
interface flavor_set_model
   module procedure flavor_set_model_single
   module procedure flavor_set_model_array
end interface
```

⟨*Flavors: procedures*⟩+≡

```
subroutine flavor_set_model_single (flv, model)
  type(flavor_t), intent(inout) :: flv
  type(model_t), intent(in), target :: model
  if (flv%f /= UNDEFINED) &
       flv%prt => model_get_particle_ptr (model, flv%f)
end subroutine flavor_set_model_single

subroutine flavor_set_model_array (flv, model)
  type(flavor_t), dimension(:), intent(inout) :: flv
  type(model_t), intent(in), target :: model
  integer :: i
  do i = 1, size (flv)
     if (flv(i)%f /= UNDEFINED) &
          flv(i)%prt => model_get_particle_ptr (model, flv(i)%f)
  end do
end subroutine flavor_set_model_array
```

Predicates

Return the definition status

⟨*Flavors: public*⟩+≡

```
public :: flavor_is_defined
```

⟨*Flavors: procedures*⟩+≡

```
elemental function flavor_is_defined (flv) result (defined)
  logical :: defined
  type(flavor_t), intent(in) :: flv
  defined = flv%f /= UNDEFINED
end function flavor_is_defined
```

Check for valid flavor (including undefined):

⟨*Flavors: public*⟩+≡

```
public :: flavor_is_valid
```

⟨*Flavors: procedures*⟩+≡

```
elemental function flavor_is_valid (flv) result (valid)
  logical :: valid
  type(flavor_t), intent(in) :: flv
  valid = flv%f /= INVALID
end function flavor_is_valid
```

Return true if the particle-data pointer is associated. (Debugging aid)

⟨*Flavors: public*⟩+≡

```
public :: flavor_is_associated
```

⟨*Flavors: procedures*⟩+≡
```
    elemental function flavor_is_associated (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
      flag = associated (flv%prt)
    end function flavor_is_associated
```

Accessing contents

With the exception of the PDG code, all particle properties are accessible only via the `prt` pointer. If this is unassigned, some access function will crash.

Return the flavor as an integer

⟨*Flavors: public*⟩+≡
```
    public :: flavor_get_pdg
```

⟨*Flavors: procedures*⟩+≡
```
    elemental function flavor_get_pdg (flv) result (f)
      integer :: f
      type(flavor_t), intent(in) :: flv
      f = flv%f
    end function flavor_get_pdg
```

Return the flavor of the antiparticle

⟨*Flavors: public*⟩+≡
```
    public :: flavor_get_pdg_anti
```

⟨*Flavors: procedures*⟩+≡
```
    elemental function flavor_get_pdg_anti (flv) result (f)
      integer :: f
      type(flavor_t), intent(in) :: flv
      if (particle_data_has_antiparticle (flv%prt)) then
         f = -flv%f
      else
         f = flv%f
      end if
    end function flavor_get_pdg_anti
```

Absolute value:

⟨*Flavors: public*⟩+≡
```
    public :: flavor_get_pdg_abs
```

⟨*Flavors: procedures*⟩+≡
```
    elemental function flavor_get_pdg_abs (flv) result (f)
      integer :: f
      type(flavor_t), intent(in) :: flv
      f = abs (flv%f)
    end function flavor_get_pdg_abs
```

Generic properties

⟨*Flavors: public*⟩+≡
```
    public :: flavor_is_visible
    public :: flavor_is_parton
```

```
      public :: flavor_is_beam_remnant
      public :: flavor_is_gauge
      public :: flavor_is_left_handed
      public :: flavor_is_right_handed
      public :: flavor_is_antiparticle
      public :: flavor_has_antiparticle
      public :: flavor_is_stable
      public :: flavor_decays_isotropically
      public :: flavor_decays_diagonal
```

⟨*Flavors: procedures*⟩+≡

```
    elemental function flavor_is_visible (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
      flag = particle_data_is_visible (flv%prt)
    end function flavor_is_visible

    elemental function flavor_is_parton (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
      flag = particle_data_is_parton (flv%prt)
    end function flavor_is_parton

    elemental function flavor_is_beam_remnant (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
      select case (abs (flv%f))
      case (HADRON_REMNANT, &
            HADRON_REMNANT_SINGLET, HADRON_REMNANT_TRIPLET, HADRON_REMNANT_OCTET)
         flag = .true.
      case default
         flag = .false.
      end select
    end function flavor_is_beam_remnant

    elemental function flavor_is_gauge (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
      flag = particle_data_is_gauge (flv%prt)
    end function flavor_is_gauge

    elemental function flavor_is_left_handed (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
      if (flv%f > 0) then
         flag = particle_data_is_left_handed (flv%prt)
      else
         flag = particle_data_is_right_handed (flv%prt)
      end if
    end function flavor_is_left_handed

    elemental function flavor_is_right_handed (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
      if (flv%f > 0) then
```

```
        flag = particle_data_is_right_handed (flv%prt)
      else
        flag = particle_data_is_left_handed (flv%prt)
      end if
    end function flavor_is_right_handed

    elemental function flavor_is_antiparticle (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
      flag = flv%f < 0
    end function flavor_is_antiparticle

    elemental function flavor_has_antiparticle (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
      flag = particle_data_has_antiparticle (flv%prt)
    end function flavor_has_antiparticle

    elemental function flavor_is_stable (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
      flag = particle_data_is_stable (flv%prt, anti = flv%f < 0)
    end function flavor_is_stable

    elemental function flavor_decays_isotropically (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
      flag = particle_data_decays_isotropically (flv%prt, anti = flv%f < 0)
    end function flavor_decays_isotropically

    elemental function flavor_decays_diagonal (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
      flag = particle_data_decays_diagonal (flv%prt, anti = flv%f < 0)
    end function flavor_decays_diagonal
```

Names:

⟨*Flavors: public*⟩+≡

```
    public :: flavor_get_name
    public :: flavor_get_tex_name
```

⟨*Flavors: procedures*⟩+≡

```
    elemental function flavor_get_name (flv) result (name)
      type(string_t) :: name
      type(flavor_t), intent(in) :: flv
      if (associated (flv%prt)) then
        name = particle_data_get_name (flv%prt, flv%f < 0)
      else
        name = "?"
      end if
    end function flavor_get_name

    elemental function flavor_get_tex_name (flv) result (name)
      type(string_t) :: name
```

```
      type(flavor_t), intent(in) :: flv
      if (associated (flv%prt)) then
         name = particle_data_get_tex_name (flv%prt, flv%f < 0)
      else
         name = "?"
      end if
   end function flavor_get_tex_name
```

⟨*Flavors: public*⟩+≡
```
   public :: flavor_get_spin_type
   public :: flavor_get_multiplicity
   public :: flavor_get_isospin_type
   public :: flavor_get_charge_type
   public :: flavor_get_color_type
```
⟨*Flavors: procedures*⟩+≡
```
   elemental function flavor_get_spin_type (flv) result (type)
     integer :: type
     type(flavor_t), intent(in) :: flv
     type = particle_data_get_spin_type (flv%prt)
   end function flavor_get_spin_type

   elemental function flavor_get_multiplicity (flv) result (type)
     integer :: type
     type(flavor_t), intent(in) :: flv
     type = particle_data_get_multiplicity (flv%prt)
   end function flavor_get_multiplicity

   elemental function flavor_get_isospin_type (flv) result (type)
     integer :: type
     type(flavor_t), intent(in) :: flv
     type = particle_data_get_isospin_type (flv%prt)
   end function flavor_get_isospin_type

   elemental function flavor_get_charge_type (flv) result (type)
     integer :: type
     type(flavor_t), intent(in) :: flv
     type = particle_data_get_charge_type (flv%prt)
   end function flavor_get_charge_type

   elemental function flavor_get_color_type (flv) result (type)
     integer :: type
     type(flavor_t), intent(in) :: flv
     if (flavor_is_antiparticle (flv)) then
        type = - particle_data_get_color_type (flv%prt)
     else
        type = particle_data_get_color_type (flv%prt)
     end if
   end function flavor_get_color_type
```

These functions return real values:

⟨*Flavors: public*⟩+≡
```
   public :: flavor_get_charge
   public :: flavor_get_mass
```

```
      public :: flavor_get_width
      public :: flavor_get_isospin
⟨Flavors: procedures⟩+≡
    elemental function flavor_get_charge (flv) result (charge)
      real(default) :: charge
      type(flavor_t), intent(in) :: flv
      if (associated (flv%prt)) then
         if (flavor_is_antiparticle (flv)) then
            charge = particle_data_get_charge (flv%prt)
         else
            charge = - particle_data_get_charge (flv%prt)
         end if
      else
         charge = 0
      end if
    end function flavor_get_charge

    elemental function flavor_get_mass (flv) result (mass)
      real(default) :: mass
      type(flavor_t), intent(in) :: flv
      if (associated (flv%prt)) then
         mass = particle_data_get_mass (flv%prt)
      else
         mass = 0
      end if
    end function flavor_get_mass

    elemental function flavor_get_width (flv) result (width)
      real(default) :: width
      type(flavor_t), intent(in) :: flv
      if (associated (flv%prt)) then
         width = particle_data_get_width (flv%prt)
      else
         width = 0
      end if
    end function flavor_get_width

    elemental function flavor_get_isospin (flv) result (isospin)
      real(default) :: isospin
      type(flavor_t), intent(in) :: flv
      if (associated (flv%prt)) then
         if (flavor_is_antiparticle (flv)) then
            isospin = particle_data_get_isospin (flv%prt)
         else
            isospin = - particle_data_get_isospin (flv%prt)
         end if
      else
         isospin = 0
      end if
    end function flavor_get_isospin
```

Comparisons

If one of the flavors is undefined, the other defined, they match.

⟨*Flavors: public*⟩+≡
```
    public :: operator(.match.)
    public :: operator(==)
    public :: operator(/=)
```

⟨*Flavors: interfaces*⟩+≡
```
    interface operator(.match.)
       module procedure flavor_match
    end interface
    interface operator(==)
       module procedure flavor_eq
    end interface
    interface operator(/=)
       module procedure flavor_neq
    end interface
```

⟨*Flavors: procedures*⟩+≡
```
    elemental function flavor_match (flv1, flv2) result (eq)
      logical :: eq
      type(flavor_t), intent(in) :: flv1, flv2
      if (flv1%f /= UNDEFINED .and. flv2%f /= UNDEFINED) then
         eq = flv1%f == flv2%f
      else
         eq = .true.
      end if
    end function flavor_match

    elemental function flavor_eq (flv1, flv2) result (eq)
      logical :: eq
      type(flavor_t), intent(in) :: flv1, flv2
      if (flv1%f /= UNDEFINED .and. flv2%f /= UNDEFINED) then
         eq = flv1%f == flv2%f
      else if (flv1%f == UNDEFINED .and. flv2%f == UNDEFINED) then
         eq = .true.
      else
         eq = .false.
      end if
    end function flavor_eq
```

⟨*Flavors: procedures*⟩+≡
```
    elemental function flavor_neq (flv1, flv2) result (neq)
      logical :: neq
      type(flavor_t), intent(in) :: flv1, flv2
      if (flv1%f /= UNDEFINED .and. flv2%f /= UNDEFINED) then
         neq = flv1%f /= flv2%f
      else if (flv1%f == UNDEFINED .and. flv2%f == UNDEFINED) then
         neq = .false.
      else
         neq = .true.
      end if
    end function flavor_neq
```

540

Tools

Merge two flavor indices. This works only if both are equal or either one is undefined, because we have no off-diagonal flavor entries. Otherwise, generate an invalid flavor.

We cannot use elemental procedures because of the pointer component.

⟨*Flavors: public*⟩+≡
```
    public :: operator(.merge.)
```

⟨*Flavors: interfaces*⟩+≡
```
    interface operator(.merge.)
       module procedure merge_flavors0
       module procedure merge_flavors1
    end interface
```

⟨*Flavors: procedures*⟩+≡
```
    function merge_flavors0 (flv1, flv2) result (flv)
      type(flavor_t) :: flv
      type(flavor_t), intent(in) :: flv1, flv2
      if (flavor_is_defined (flv1) .and. flavor_is_defined (flv2)) then
         if (flv1 == flv2) then
            flv = flv1
         else
            flv%f = INVALID
         end if
      else if (flavor_is_defined (flv1)) then
         flv = flv1
      else if (flavor_is_defined (flv2)) then
         flv = flv2
      end if
    end function merge_flavors0

    function merge_flavors1 (flv1, flv2) result (flv)
      type(flavor_t), dimension(:), intent(in) :: flv1, flv2
      type(flavor_t), dimension(size(flv1)) :: flv
      integer :: i
      do i = 1, size (flv1)
         flv(i) = flv1(i) .merge. flv2(i)
      end do
    end function merge_flavors1
```

Generate consecutive color indices for a given flavor. The indices are counted starting with the stored value of c, so new indices are created each time this (impure) function is called. The counter can be reset by the optional argument `c_seed` if desired. The optional flag `reverse` is used only for octets. If set, the color and anticolor entries of the octet particle are exchanged.

⟨*Flavors: public*⟩+≡
```
    public :: color_from_flavor
```

⟨*Flavors: interfaces*⟩+≡
```
    interface color_from_flavor
       module procedure color_from_flavor0
       module procedure color_from_flavor1
    end interface
```

⟨*Flavors: procedures*⟩+≡

```
function color_from_flavor0 (flv, c_seed, reverse) result (col)
  type(color_t) :: col
  type(flavor_t), intent(in) :: flv
  integer, intent(in), optional :: c_seed
  logical, intent(in), optional :: reverse
  integer, save :: c = 1
  logical :: rev
  if (present (c_seed))  c = c_seed
  rev = .false.;  if (present (reverse)) rev = reverse
  select case (flavor_get_color_type (flv))
  case (1)
  case (3)
     call color_init (col, (/ c /));  c = c + 1
  case (-3)
     call color_init (col, (/-c /));  c = c + 1
  case (8)
     if (rev) then
        call color_init (col, (/ c+1, -c /));  c = c + 2
     else
        call color_init (col, (/ c, -(c+1) /));  c = c + 2
     end if
  end select
end function color_from_flavor0

function color_from_flavor1 (flv, c_seed, reverse) result (col)
  type(flavor_t), dimension(:), intent(in) :: flv
  integer, intent(in), optional :: c_seed
  logical, intent(in), optional :: reverse
  type(color_t), dimension(size(flv)) :: col
  integer :: i
  col(1) = color_from_flavor0 (flv(1), c_seed, reverse)
  do i = 2, size (flv)
     col(i) = color_from_flavor0 (flv(i), reverse=reverse)
  end do
end function color_from_flavor1
```

This procedure returns the flavor object for the antiparticle. The antiparticle code may either be the same code or its negative.

⟨*Flavors: public*⟩+≡

```
public :: flavor_anti
```

⟨*Flavors: procedures*⟩+≡

```
function flavor_anti (flv) result (aflv)
  type(flavor_t) :: aflv
  type(flavor_t), intent(in) :: flv
  if (flavor_has_antiparticle (flv)) then
     aflv%f = - flv%f
  else
     aflv%f = flv%f
  end if
  aflv%prt => flv%prt
end function flavor_anti
```

7.4 Quantum numbers

This module collects helicity, color, and flavor in a single type and defines procedures

⟨`quantum_numbers.f90`⟩≡
  ⟨*File header*⟩

  ```
  module quantum_numbers
  ```

  ⟨*Use file utils*⟩
  ```
    use models
    use flavors
    use colors
    use helicities
  ```

  ⟨*Standard module head*⟩

  ⟨*Quantum numbers: public*⟩

  ⟨*Quantum numbers: types*⟩

  ⟨*Quantum numbers: interfaces*⟩

  ```
  contains
  ```

  ⟨*Quantum numbers: procedures*⟩

  ```
  end module quantum_numbers
  ```

7.4.1 The quantum number type

⟨*Quantum numbers: public*⟩≡
  ```
    public :: quantum_numbers_t
  ```

⟨*Quantum numbers: types*⟩≡
  ```
    type :: quantum_numbers_t
      private
      type(flavor_t) :: f
      type(color_t) :: c
      type(helicity_t) :: h
    end type quantum_numbers_t
  ```

Define quantum numbers: Initializer form. All arguments may be present or absent.

⟨*Quantum numbers: public*⟩+≡
  ```
    public :: quantum_numbers_init
  ```

⟨*Quantum numbers: interfaces*⟩≡
  ```
    interface quantum_numbers_init
      module procedure quantum_numbers_init0_f
      module procedure quantum_numbers_init0_c
      module procedure quantum_numbers_init0_h
      module procedure quantum_numbers_init0_fc
      module procedure quantum_numbers_init0_fh
  ```

```
      module procedure quantum_numbers_init0_ch
      module procedure quantum_numbers_init0_fch
      module procedure quantum_numbers_init1_f
      module procedure quantum_numbers_init1_c
      module procedure quantum_numbers_init1_h
      module procedure quantum_numbers_init1_fc
      module procedure quantum_numbers_init1_fh
      module procedure quantum_numbers_init1_ch
      module procedure quantum_numbers_init1_fch
    end interface
```

⟨*Quantum numbers: procedures*⟩≡
```
    subroutine quantum_numbers_init0_f (qn, flv)
      type(quantum_numbers_t), intent(out) :: qn
      type(flavor_t), intent(in) :: flv
      qn%f = flv
    end subroutine quantum_numbers_init0_f

    subroutine quantum_numbers_init0_c (qn, col)
      type(quantum_numbers_t), intent(out) :: qn
      type(color_t), intent(in) :: col
      qn%c = col
    end subroutine quantum_numbers_init0_c

    subroutine quantum_numbers_init0_h (qn, hel)
      type(quantum_numbers_t), intent(out) :: qn
      type(helicity_t), intent(in) :: hel
      qn%h = hel
    end subroutine quantum_numbers_init0_h

    subroutine quantum_numbers_init0_fc (qn, flv, col)
      type(quantum_numbers_t), intent(out) :: qn
      type(flavor_t), intent(in) :: flv
      type(color_t), intent(in) :: col
      qn%f = flv
      qn%c = col
    end subroutine quantum_numbers_init0_fc

    subroutine quantum_numbers_init0_fh (qn, flv, hel)
      type(quantum_numbers_t), intent(out) :: qn
      type(flavor_t), intent(in) :: flv
      type(helicity_t), intent(in) :: hel
      qn%f = flv
      qn%h = hel
    end subroutine quantum_numbers_init0_fh

    subroutine quantum_numbers_init0_ch (qn, col, hel)
      type(quantum_numbers_t), intent(out) :: qn
      type(color_t), intent(in) :: col
      type(helicity_t), intent(in) :: hel
      qn%c = col
      qn%h = hel
    end subroutine quantum_numbers_init0_ch
```

```
subroutine quantum_numbers_init0_fch (qn, flv, col, hel)
  type(quantum_numbers_t), intent(out) :: qn
  type(flavor_t), intent(in) :: flv
  type(color_t), intent(in) :: col
  type(helicity_t), intent(in) :: hel
  qn%f = flv
  qn%c = col
  qn%h = hel
end subroutine quantum_numbers_init0_fch

subroutine quantum_numbers_init1_f (qn, flv)
  type(quantum_numbers_t), dimension(:), intent(out) :: qn
  type(flavor_t), dimension(:), intent(in) :: flv
  integer :: i
  do i = 1, size (qn)
     call quantum_numbers_init0_f (qn(i), flv(i))
  end do
end subroutine quantum_numbers_init1_f

subroutine quantum_numbers_init1_c (qn, col)
  type(quantum_numbers_t), dimension(:), intent(out) :: qn
  type(color_t), dimension(:), intent(in) :: col
  integer :: i
  do i = 1, size (qn)
     call quantum_numbers_init0_c (qn(i), col(i))
  end do
end subroutine quantum_numbers_init1_c

subroutine quantum_numbers_init1_h (qn, hel)
  type(quantum_numbers_t), dimension(:), intent(out) :: qn
  type(helicity_t), dimension(:), intent(in) :: hel
  integer :: i
  do i = 1, size (qn)
     call quantum_numbers_init0_h (qn(i), hel(i))
  end do
end subroutine quantum_numbers_init1_h

subroutine quantum_numbers_init1_fc (qn, flv, col)
  type(quantum_numbers_t), dimension(:), intent(out) :: qn
  type(flavor_t), dimension(:), intent(in) :: flv
  type(color_t), dimension(:), intent(in) :: col
  integer :: i
  do i = 1, size (qn)
     call quantum_numbers_init0_fc (qn(i), flv(i), col(i))
  end do
end subroutine quantum_numbers_init1_fc

subroutine quantum_numbers_init1_fh (qn, flv, hel)
  type(quantum_numbers_t), dimension(:), intent(out) :: qn
  type(flavor_t), dimension(:), intent(in) :: flv
  type(helicity_t), dimension(:), intent(in) :: hel
  integer :: i
  do i = 1, size (qn)
     call quantum_numbers_init0_fh (qn(i), flv(i), hel(i))
```

```
      end do
    end subroutine quantum_numbers_init1_fh

    subroutine quantum_numbers_init1_ch (qn, col, hel)
      type(quantum_numbers_t), dimension(:), intent(out) :: qn
      type(color_t), dimension(:), intent(in) :: col
      type(helicity_t), dimension(:), intent(in) :: hel
      integer :: i
      do i = 1, size (qn)
        call quantum_numbers_init0_ch (qn(i), col(i), hel(i))
      end do
    end subroutine quantum_numbers_init1_ch

    subroutine quantum_numbers_init1_fch (qn, flv, col, hel)
      type(quantum_numbers_t), dimension(:), intent(out) :: qn
      type(flavor_t), dimension(:), intent(in) :: flv
      type(color_t), dimension(:), intent(in) :: col
      type(helicity_t), dimension(:), intent(in) :: hel
      integer :: i
      do i = 1, size (qn)
        call quantum_numbers_init0_fch (qn(i), flv(i), col(i), hel(i))
      end do
    end subroutine quantum_numbers_init1_fch
```

7.4.2 I/O

Write the quantum numbers in condensed form, enclosed by square brackets.
For convenience, introduce also an array version.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_write
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface quantum_numbers_write
       module procedure quantum_numbers_write_single
       module procedure quantum_numbers_write_array
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    subroutine quantum_numbers_write_single (qn, unit)
      type(quantum_numbers_t), intent(in) :: qn
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)", advance="no")  "["
      if (flavor_is_defined (qn%f)) then
         call flavor_write (qn%f, u)
         if (color_is_defined (qn%c) .or. helicity_is_defined (qn%h)) &
              write (u, "(1x)", advance="no")
      end if
      if (color_is_defined (qn%c) .or. color_is_ghost (qn%c)) then
         call color_write (qn%c, u)
         if (helicity_is_defined (qn%h))  write (u, "(1x)", advance="no")
      end if
```

```
      if (helicity_is_defined (qn%h)) then
         call helicity_write (qn%h, u)
      end if
      write (u, "(A)", advance="no")  "]"
   end subroutine quantum_numbers_write_single

   subroutine quantum_numbers_write_array (qn, unit)
     type(quantum_numbers_t), dimension(:), intent(in) :: qn
     integer, intent(in), optional :: unit
     integer :: i
     integer :: u
     u = output_unit (unit);  if (u < 0)  return
     write (u, "(A)", advance="no")  "["
     do i = 1, size (qn)
        if (i > 1)  write (u, "(A)", advance="no")  " / "
        if (flavor_is_defined (qn(i)%f)) then
           call flavor_write (qn(i)%f, u)
           if (color_is_defined (qn(i)%c) .or. helicity_is_defined (qn(i)%h)) &
                write (u, "(1x)", advance="no")
        end if
        if (color_is_defined (qn(i)%c) .or. color_is_ghost (qn(i)%c)) then
           call color_write (qn(i)%c, u)
           if (helicity_is_defined (qn(i)%h))  write (u, "(1x)", advance="no")
        end if
        if (helicity_is_defined (qn(i)%h)) then
           call helicity_write (qn(i)%h, u)
        end if
     end do
     write (u, "(A)", advance="no")  "]"
   end subroutine quantum_numbers_write_array
```

Binary I/O.

⟨*Quantum numbers: public*⟩+≡
```
   public :: quantum_numbers_write_raw
   public :: quantum_numbers_read_raw
```

⟨*Quantum numbers: procedures*⟩+≡
```
   subroutine quantum_numbers_write_raw (qn, u)
     type(quantum_numbers_t), intent(in) :: qn
     integer, intent(in) :: u
     call flavor_write_raw (qn%f, u)
     call color_write_raw (qn%c, u)
     call helicity_write_raw (qn%h, u)
   end subroutine quantum_numbers_write_raw

   subroutine quantum_numbers_read_raw (qn, u, iostat)
     type(quantum_numbers_t), intent(out) :: qn
     integer, intent(in) :: u
     integer, intent(out), optional :: iostat
     call flavor_read_raw (qn%f, u, iostat=iostat)
     call color_read_raw (qn%c, u, iostat=iostat)
     call helicity_read_raw (qn%h, u, iostat=iostat)
   end subroutine quantum_numbers_read_raw
```

7.4.3 Accessing contents

Color and helicity can be done by elemental functions. Flavor needs explicit specifics because of the pointer assignment.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_get_flavor
    public :: quantum_numbers_get_color
    public :: quantum_numbers_get_helicity
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface quantum_numbers_get_flavor
       module procedure quantum_numbers_get_flavor0
       module procedure quantum_numbers_get_flavor1
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    function quantum_numbers_get_flavor0 (qn) result (flv)
      type(flavor_t) :: flv
      type(quantum_numbers_t), intent(in) :: qn
      flv = qn%f
    end function quantum_numbers_get_flavor0

    function quantum_numbers_get_flavor1 (qn) result (flv)
      type(quantum_numbers_t), dimension(:), intent(in) :: qn
      type(flavor_t), dimension(size(qn)) :: flv
      integer :: i
      do i = 1, size (qn)
         flv(i) = qn(i)%f
      end do
    end function quantum_numbers_get_flavor1

    elemental function quantum_numbers_get_color (qn) result (col)
      type(color_t) :: col
      type(quantum_numbers_t), intent(in) :: qn
      col = qn%c
    end function quantum_numbers_get_color

    elemental function quantum_numbers_get_helicity (qn) result (hel)
      type(helicity_t) :: hel
      type(quantum_numbers_t), intent(in) :: qn
      hel = qn%h
    end function quantum_numbers_get_helicity
```

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_set_flavor
    public :: quantum_numbers_set_color
    public :: quantum_numbers_set_helicity
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface quantum_numbers_set_flavor
       module procedure quantum_numbers_set_flavor0
       module procedure quantum_numbers_set_flavor1
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    subroutine quantum_numbers_set_flavor0 (qn, flv)
      type(quantum_numbers_t), intent(inout) :: qn
      type(flavor_t), intent(in) :: flv
      qn%f = flv
    end subroutine quantum_numbers_set_flavor0

    subroutine quantum_numbers_set_flavor1 (qn, flv)
      type(quantum_numbers_t), dimension(:), intent(inout) :: qn
      type(flavor_t), dimension(:), intent(in) :: flv
      integer :: i
      do i = 1, size (flv)
         qn(i)%f = flv(i)
      end do
    end subroutine quantum_numbers_set_flavor1

    elemental subroutine quantum_numbers_set_color (qn, col)
      type(quantum_numbers_t), intent(inout) :: qn
      type(color_t), intent(in) :: col
      qn%c = col
    end subroutine quantum_numbers_set_color

    elemental subroutine quantum_numbers_set_helicity (qn, hel)
      type(quantum_numbers_t), intent(inout) :: qn
      type(helicity_t), intent(in) :: hel
      qn%h = hel
    end subroutine quantum_numbers_set_helicity
```

This just resets the ghost property of the color/helicity part:

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_set_color_ghost
    public :: quantum_numbers_set_helicity_ghost
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental subroutine quantum_numbers_set_color_ghost (qn, ghost)
      type(quantum_numbers_t), intent(inout) :: qn
      logical, intent(in) :: ghost
      call color_set_ghost (qn%c, ghost)
    end subroutine quantum_numbers_set_color_ghost

    elemental subroutine quantum_numbers_set_helicity_ghost (qn, ghost)
      type(quantum_numbers_t), intent(inout) :: qn
      logical, intent(in) :: ghost
      call helicity_set_ghost (qn%h, ghost)
    end subroutine quantum_numbers_set_helicity_ghost
```

Assign a model to the flavor part of quantum numbers.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_set_model
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface quantum_numbers_set_model
       module procedure quantum_numbers_set_model_single
       module procedure quantum_numbers_set_model_array
```

549

```
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    subroutine quantum_numbers_set_model_single (qn, model)
      type(quantum_numbers_t), intent(inout) :: qn
      type(model_t), intent(in), target :: model
      call flavor_set_model (qn%f, model)
    end subroutine quantum_numbers_set_model_single

    subroutine quantum_numbers_set_model_array (qn, model)
      type(quantum_numbers_t), dimension(:), intent(inout) :: qn
      type(model_t), intent(in), target :: model
      call flavor_set_model (qn%f, model)
    end subroutine quantum_numbers_set_model_array
```

This is a convenience function: return the color type for the flavor (array).

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_get_color_type
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_get_color_type (qn) result (color_type)
      integer :: color_type
      type(quantum_numbers_t), intent(in) :: qn
      color_type = flavor_get_color_type (qn%f)
    end function quantum_numbers_get_color_type
```

7.4.4 Predicates

Check if the flavor index is valid (including UNDEFINED).

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_are_valid
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_are_valid (qn) result (valid)
      logical :: valid
      type(quantum_numbers_t), intent(in) :: qn
      valid = flavor_is_valid (qn%f)
    end function quantum_numbers_are_valid
```

Check if the flavor part has its particle-data pointer associated (debugging aid).

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_are_associated
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_are_associated (qn) result (flag)
      logical :: flag
      type(quantum_numbers_t), intent(in) :: qn
      flag = flavor_is_associated (qn%f)
    end function quantum_numbers_are_associated
```

Check if the helicity and color quantum numbers are diagonal. (Unpolarized/colorless also counts as diagonal.) Flavor is diagonal by definition.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_are_diagonal
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_are_diagonal (qn) result (diagonal)
      logical :: diagonal
      type(quantum_numbers_t), intent(in) :: qn
      diagonal = helicity_is_diagonal (qn%h) .and. color_is_diagonal (qn%c)
    end function quantum_numbers_are_diagonal
```

Check if the color and/or helicity part has the ghost property.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_is_color_ghost
    public :: quantum_numbers_is_helicity_ghost
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_is_color_ghost (qn) result (ghost)
      logical :: ghost
      type(quantum_numbers_t), intent(in) :: qn
      ghost = color_is_ghost (qn%c)
    end function quantum_numbers_is_color_ghost

    elemental function quantum_numbers_is_helicity_ghost (qn) result (ghost)
      logical :: ghost
      type(quantum_numbers_t), intent(in) :: qn
      ghost = helicity_is_ghost (qn%h)
    end function quantum_numbers_is_helicity_ghost
```

7.4.5 Comparisons

Matching and equality is derived from the individual quantum numbers. The variant `fhmatch` matches only flavor and helicity. The variant `dhmatch` matches only diagonal helicity, if the matching helicity is undefined.

⟨*Quantum numbers: public*⟩+≡
```
    public :: operator(.match.)
    public :: operator(.fmatch.)
    public :: operator(.fhmatch.)
    public :: operator(.dhmatch.)
    public :: operator(==)
    public :: operator(/=)
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface operator(.match.)
       module procedure quantum_numbers_match
    end interface
    interface operator(.fmatch.)
       module procedure quantum_numbers_match_f
    end interface
    interface operator(.fhmatch.)
       module procedure quantum_numbers_match_fh
    end interface
```

```
      interface operator(.dhmatch.)
         module procedure quantum_numbers_match_hel_diag
      end interface
      interface operator(==)
         module procedure quantum_numbers_eq
      end interface
      interface operator(/=)
         module procedure quantum_numbers_neq
      end interface
```

⟨*Quantum numbers: procedures*⟩+≡

```
      elemental function quantum_numbers_match (qn1, qn2) result (match)
        logical :: match
        type(quantum_numbers_t), intent(in) :: qn1, qn2
        match = (qn1%f .match. qn2%f) .and. &
             (qn1%c .match. qn2%c) .and. &
             (qn1%h .match. qn2%h)
      end function quantum_numbers_match

      elemental function quantum_numbers_match_f (qn1, qn2) result (match)
        logical :: match
        type(quantum_numbers_t), intent(in) :: qn1, qn2
        match = (qn1%f .match. qn2%f)
      end function quantum_numbers_match_f

      elemental function quantum_numbers_match_fh (qn1, qn2) result (match)
        logical :: match
        type(quantum_numbers_t), intent(in) :: qn1, qn2
        match = (qn1%f .match. qn2%f) .and. &
             (qn1%h .match. qn2%h)
      end function quantum_numbers_match_fh

      elemental function quantum_numbers_match_hel_diag (qn1, qn2) result (match)
        logical :: match
        type(quantum_numbers_t), intent(in) :: qn1, qn2
        match = (qn1%f .match. qn2%f) .and. &
             (qn1%c .match. qn2%c) .and. &
             (qn1%h .dmatch. qn2%h)
      end function quantum_numbers_match_hel_diag

      elemental function quantum_numbers_eq (qn1, qn2) result (eq)
        logical :: eq
        type(quantum_numbers_t), intent(in) :: qn1, qn2
        eq = (qn1%f == qn2%f) .and. &
             (qn1%c == qn2%c) .and. &
             (qn1%h == qn2%h)
      end function quantum_numbers_eq

      elemental function quantum_numbers_neq (qn1, qn2) result (neq)
        logical :: neq
        type(quantum_numbers_t), intent(in) :: qn1, qn2
        neq = (qn1%f /= qn2%f) .or. &
             (qn1%c /= qn2%c) .or. &
             (qn1%h /= qn2%h)
      end function quantum_numbers_neq
```

Two sets of quantum numbers are compatible if the individual quantum numbers are compatible, depending on the mask. Flavor has to match, regardless of the flavor mask.

If the color flag is set, color is compatible if the ghost property is identical. If the color flag is unset, color has to be identical. I.e., if the flag is set, the color amplitudes can interfere. If it is not set, they must be identical, and there must be no ghost. The latter property is used for expanding physical color flows.

Helicity is compatible if the mask is unset, otherwise it has to match. This determines if two amplitudes can be multiplied (no mask) or traced (mask).

⟨*Quantum numbers: public*⟩+≡
```
public :: quantum_numbers_are_compatible
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental function quantum_numbers_are_compatible (qn1, qn2, mask) &
   result (flag)
  logical :: flag
  type(quantum_numbers_t), intent(in) :: qn1, qn2
  type(quantum_numbers_mask_t), intent(in) :: mask
  if (mask%h .or. mask%hd) then
     flag = (qn1%f .match. qn2%f) .and. (qn1%h .match. qn2%h)
  else
     flag = (qn1%f .match. qn2%f)
  end if
  if (mask%c) then
     flag = flag .and. (color_is_ghost (qn1%c) .eqv. color_is_ghost (qn2%c))
  else
     flag = flag .and. &
          .not. (color_is_ghost (qn1%c) .or. color_is_ghost (qn2%c)) .and. &
          (qn1%c == qn2%c)
  end if
end function quantum_numbers_are_compatible
```

This is the analog for a single quantum-number set. We just check for color ghosts; they are excluded if the color mask is unset (color-flow expansion).

⟨*Quantum numbers: public*⟩+≡
```
public :: quantum_numbers_are_physical
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental function quantum_numbers_are_physical (qn, mask) result (flag)
  logical :: flag
  type(quantum_numbers_t), intent(in) :: qn
  type(quantum_numbers_mask_t), intent(in) :: mask
  if (mask%c) then
     flag = .true.
  else
     flag = .not. color_is_ghost (qn%c)
  end if
end function quantum_numbers_are_physical
```

7.4.6 Operations

Inherited from the color component: reassign color indices in canonical order.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_canonicalize_color
```

⟨*Quantum numbers: procedures*⟩+≡
```
    subroutine quantum_numbers_canonicalize_color (qn)
      type(quantum_numbers_t), dimension(:), intent(inout) :: qn
      call color_canonicalize (qn%c)
    end subroutine quantum_numbers_canonicalize_color
```

Inherited from the color component: make a color map for two matching quantum-number arrays.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_set_color_map
```

⟨*Quantum numbers: procedures*⟩+≡
```
    subroutine quantum_numbers_set_color_map (map, qn1, qn2)
      integer, dimension(:,:), intent(out), allocatable :: map
      type(quantum_numbers_t), dimension(:), intent(in) :: qn1, qn2
      call set_color_map (map, qn1%c, qn2%c)
    end subroutine quantum_numbers_set_color_map
```

Inherited from the color component: translate the color part using a color-map array

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_translate_color
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface quantum_numbers_translate_color
       module procedure quantum_numbers_translate_color0
       module procedure quantum_numbers_translate_color1
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    subroutine quantum_numbers_translate_color0 (qn, map, offset)
      type(quantum_numbers_t), intent(inout) :: qn
      integer, dimension(:,:), intent(in) :: map
      integer, intent(in), optional :: offset
      call color_translate (qn%c, map, offset)
    end subroutine quantum_numbers_translate_color0

    subroutine quantum_numbers_translate_color1 (qn, map, offset)
      type(quantum_numbers_t), dimension(:), intent(inout) :: qn
      integer, dimension(:,:), intent(in) :: map
      integer, intent(in), optional :: offset
      call color_translate (qn%c, map, offset)
    end subroutine quantum_numbers_translate_color1
```

Inherited from the color component: return the color index with highest absolute value

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_get_max_color_value
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface quantum_numbers_get_max_color_value
       module procedure quantum_numbers_get_max_color_value0
       module procedure quantum_numbers_get_max_color_value1
       module procedure quantum_numbers_get_max_color_value2
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    function quantum_numbers_get_max_color_value0 (qn) result (cmax)
      integer :: cmax
      type(quantum_numbers_t), intent(in) :: qn
      cmax = color_get_max_value0 (qn%c)
      !!! ifort 11.1 v5 demands this
      !!! cmax = color_get_max_value (qn%c)
    end function quantum_numbers_get_max_color_value0

    function quantum_numbers_get_max_color_value1 (qn) result (cmax)
      integer :: cmax
      type(quantum_numbers_t), dimension(:), intent(in) :: qn
      cmax = color_get_max_value1 (qn%c)
      !!! ifort 11.1 v5 demands this
      !!! cmax = color_get_max_value (qn%c)
    end function quantum_numbers_get_max_color_value1

    function quantum_numbers_get_max_color_value2 (qn) result (cmax)
      integer :: cmax
      type(quantum_numbers_t), dimension(:,:), intent(in) :: qn
      cmax = color_get_max_value2 (qn%c)
      !!! ifort 11.1 v5 demands this
      !!! cmax = color_get_max_value (qn%c)
    end function quantum_numbers_get_max_color_value2
```

Inherited from the color component: add an offset to the indices of the color part

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_add_color_offset
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental subroutine quantum_numbers_add_color_offset (qn, offset)
      type(quantum_numbers_t), intent(inout) :: qn
      integer, intent(in) :: offset
      call color_add_offset (qn%c, offset)
    end subroutine quantum_numbers_add_color_offset
```

Given a quantum number array, return all possible color contractions, leaving the other quantum numbers intact.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_number_array_make_color_contractions
```

⟨*Quantum numbers: procedures*⟩+≡
```
    subroutine quantum_number_array_make_color_contractions (qn_in, qn_out)
      type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
      type(quantum_numbers_t), dimension(:,:), intent(out), allocatable :: qn_out
```

```
    type(color_t), dimension(:,:), allocatable :: col
    integer :: i
    call color_array_make_contractions (qn_in%c, col)
    allocate (qn_out (size (col, 1), size (col, 2)))
    do i = 1, size (qn_out, 2)
       qn_out(:,i)%f = qn_in%f
       qn_out(:,i)%c = col(:,i)
       qn_out(:,i)%h = qn_in%h
    end do
  end subroutine quantum_number_array_make_color_contractions
```

Inherited from the color component: invert the color, switching particle/antiparticle.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_invert_color
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental subroutine quantum_numbers_invert_color (qn)
      type(quantum_numbers_t), intent(inout) :: qn
      call color_invert (qn%c)
    end subroutine quantum_numbers_invert_color
```

Merge two quantum number sets: for each entry, if both are defined, combine them to an off-diagonal entry (meaningful only if the input was diagonal). If either entry is undefined, take the defined one.

For flavor, off-diagonal entries are invalid, so both flavors must be equal, otherwise an invalid flavor is inserted.

⟨*Quantum numbers: public*⟩+≡
```
    public :: operator(.merge.)
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface operator(.merge.)
       module procedure merge_quantum_numbers0
       module procedure merge_quantum_numbers1
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    function merge_quantum_numbers0 (qn1, qn2) result (qn3)
      type(quantum_numbers_t) :: qn3
      type(quantum_numbers_t), intent(in) :: qn1, qn2
      qn3%f = qn1%f .merge. qn2%f
      qn3%c = qn1%c .merge. qn2%c
      qn3%h = qn1%h .merge. qn2%h
    end function merge_quantum_numbers0

    function merge_quantum_numbers1 (qn1, qn2) result (qn3)
      type(quantum_numbers_t), dimension(:), intent(in) :: qn1, qn2
      type(quantum_numbers_t), dimension(size(qn1)) :: qn3
      qn3%f = qn1%f .merge. qn2%f
      qn3%c = qn1%c .merge. qn2%c
      qn3%h = qn1%h .merge. qn2%h
    end function merge_quantum_numbers1
```

7.4.7 The quantum number mask

The quantum numbers mask is true for quantum numbers that should be ignored or summed over. The three mandatory entries correspond to flavor, color, and helicity, respectively.

There is an additional entry `cg`: If false, the color-ghosts property should be kept even if color is ignored. This is relevant only if `c` is set, otherwise it is always false.

The flag `hd` tells that only diagonal entries in helicity should be kept. If `h` is set, `hd` is irrelevant and will be kept `.false.`.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_mask_t
```

⟨*Quantum numbers: types*⟩+≡
```
    type :: quantum_numbers_mask_t
      private
      logical :: f = .false.
      logical :: c = .false.
      logical :: cg = .false.
      logical :: h = .false.
      logical :: hd = .false.
    end type quantum_numbers_mask_t
```

Define a quantum number mask: Constructor form

⟨*Quantum numbers: public*⟩+≡
```
    public :: new_quantum_numbers_mask
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function new_quantum_numbers_mask &
        (mask_f, mask_c, mask_h, mask_cg, mask_hd) result (mask)
      type(quantum_numbers_mask_t) :: mask
      logical, intent(in) :: mask_f, mask_c, mask_h
      logical, intent(in), optional :: mask_cg
      logical, intent(in), optional :: mask_hd
      call quantum_numbers_mask_init &
          (mask, mask_f, mask_c, mask_h, mask_cg, mask_hd)
    end function new_quantum_numbers_mask
```

Define quantum numbers: Initializer form

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_mask_init
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental subroutine quantum_numbers_mask_init &
        (mask, mask_f, mask_c, mask_h, mask_cg, mask_hd)
      type(quantum_numbers_mask_t), intent(out) :: mask
      logical, intent(in) :: mask_f, mask_c, mask_h
      logical, intent(in), optional :: mask_cg, mask_hd
      mask%f = mask_f
      mask%c = mask_c
      mask%h = mask_h
      if (present (mask_cg)) then
         if (mask%c)  mask%cg = mask_cg
      else
```

```
         mask%cg = mask_c
      end if
      if (present (mask_hd)) then
         if (.not. mask%h)  mask%hd = mask_hd
      end if
    end subroutine quantum_numbers_mask_init
```

Write a quantum numbers mask

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_mask_write
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface quantum_numbers_mask_write
       module procedure quantum_numbers_mask_write_single
       module procedure quantum_numbers_mask_write_array
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    subroutine quantum_numbers_mask_write_single (mask, unit)
      type(quantum_numbers_mask_t), intent(in) :: mask
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)", advance="no") "["
      write (u, "(L1)", advance="no")  mask%f
      write (u, "(L1)", advance="no")  mask%c
      if (.not.mask%cg)  write (u, "('g')", advance="no")
      write (u, "(L1)", advance="no")  mask%h
      if (mask%hd)  write (u, "('d')", advance="no")
      write (u, "(A)", advance="no") "]"
    end subroutine quantum_numbers_mask_write_single

    subroutine quantum_numbers_mask_write_array (mask, unit)
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)", advance="no") "["
      do i = 1, size (mask)
         if (i > 1)  write (u, "(A)", advance="no")  "/"
         write (u, "(L1)", advance="no")  mask(i)%f
         write (u, "(L1)", advance="no")  mask(i)%c
         if (.not.mask(i)%cg)  write (u, "('g')", advance="no")
         write (u, "(L1)", advance="no")  mask(i)%h
         if (mask(i)%hd)  write (u, "('d')", advance="no")
      end do
      write (u, "(A)", advance="no") "]"
    end subroutine quantum_numbers_mask_write_array
```

7.4.8 Setting mask components

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_mask_set_flavor
```

```
      public :: quantum_numbers_mask_set_color
      public :: quantum_numbers_mask_set_helicity
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental subroutine quantum_numbers_mask_set_flavor (mask, mask_f)
      type(quantum_numbers_mask_t), intent(inout) :: mask
      logical, intent(in) :: mask_f
      mask%f = mask_f
    end subroutine quantum_numbers_mask_set_flavor

    elemental subroutine quantum_numbers_mask_set_color (mask, mask_c, mask_cg)
      type(quantum_numbers_mask_t), intent(inout) :: mask
      logical, intent(in) :: mask_c
      logical, intent(in), optional :: mask_cg
      mask%c = mask_c
      if (present (mask_cg)) then
         if (mask%c)  mask%cg = mask_cg
      else
         mask%cg = mask_c
      end if
    end subroutine quantum_numbers_mask_set_color

    elemental subroutine quantum_numbers_mask_set_helicity (mask, mask_h, mask_hd)
      type(quantum_numbers_mask_t), intent(inout) :: mask
      logical, intent(in) :: mask_h
      logical, intent(in), optional :: mask_hd
      mask%h = mask_h
      if (present (mask_hd)) then
         if (.not. mask%h)  mask%hd = mask_hd
      end if
    end subroutine quantum_numbers_mask_set_helicity
```

7.4.9 Mask predicates

Return true if either one of the entries is set:

⟨*Quantum numbers: public*⟩+≡
```
    public :: any
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface any
       module procedure quantum_numbers_mask_any
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    function quantum_numbers_mask_any (mask) result (match)
      logical :: match
      type(quantum_numbers_mask_t), intent(in) :: mask
      match = mask%f .or. mask%c .or. mask%h .or. mask%hd
    end function quantum_numbers_mask_any
```

7.4.10 Operators

The OR operation is applied to all components.

⟨*Quantum numbers: public*⟩+≡
```
public :: operator(.or.)
```

⟨*Quantum numbers: interfaces*⟩+≡
```
interface operator(.or.)
   module procedure quantum_numbers_mask_or
end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental function quantum_numbers_mask_or (mask1, mask2) result (mask)
   type(quantum_numbers_mask_t) :: mask
   type(quantum_numbers_mask_t), intent(in) :: mask1, mask2
   mask%f = mask1%f .or. mask2%f
   mask%c = mask1%c .or. mask2%c
   if (mask%c)  mask%cg = mask1%cg .or. mask2%cg
   mask%h = mask1%h .or. mask2%h
   if (.not. mask%h)  mask%hd = mask1%hd .or. mask2%hd
 end function quantum_numbers_mask_or
```

7.4.11 Mask comparisons

Return true if the two masks are equivalent / differ:

⟨*Quantum numbers: public*⟩+≡
```
public :: operator(.eqv.)
public :: operator(.neqv.)
```

⟨*Quantum numbers: interfaces*⟩+≡
```
interface operator(.eqv.)
   module procedure quantum_numbers_mask_eqv
end interface
interface operator(.neqv.)
   module procedure quantum_numbers_mask_neqv
end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental function quantum_numbers_mask_eqv (mask1, mask2) result (eqv)
  logical :: eqv
  type(quantum_numbers_mask_t), intent(in) :: mask1, mask2
  eqv = (mask1%f .eqv. mask2%f) .and. &
        (mask1%c .eqv. mask2%c) .and. &
        (mask1%cg .eqv. mask2%cg) .and. &
        (mask1%h .eqv. mask2%h) .and. &
        (mask1%hd .eqv. mask2%hd)
end function quantum_numbers_mask_eqv

elemental function quantum_numbers_mask_neqv (mask1, mask2) result (neqv)
  logical :: neqv
  type(quantum_numbers_mask_t), intent(in) :: mask1, mask2
  neqv = (mask1%f .neqv. mask2%f) .or. &
        (mask1%c .neqv. mask2%c) .or. &
        (mask1%cg .neqv. mask2%cg) .or. &
```

```
        (mask1%h .neqv. mask2%h) .or. &
        (mask1%hd .neqv. mask2%hd)
  end function quantum_numbers_mask_neqv
```

7.4.12 Apply a mask

Applying a mask to the quantum number object means undefining those entries
where the mask is set. The others remain unaffected.

The **hd** mask has the special property that it "diagonalizes" helicity, i.e., the
second helicity entry is dropped and the result is a diagonal helicity quantum
number.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_undefine
    public :: quantum_numbers_undefined
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface quantum_numbers_undefined
       module procedure quantum_numbers_undefined0
       module procedure quantum_numbers_undefined1
       module procedure quantum_numbers_undefined11
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental subroutine quantum_numbers_undefine (qn, mask)
      type(quantum_numbers_t), intent(inout) :: qn
      type(quantum_numbers_mask_t), intent(in) :: mask
      if (mask%f)  call flavor_undefine (qn%f)
      if (mask%c)  call color_undefine (qn%c, undefine_ghost=mask%cg)
      if (mask%h) then
         call helicity_undefine (qn%h)
      else if (mask%hd) then
         if (.not. helicity_is_diagonal (qn%h)) then
            call helicity_diagonalize (qn%h)
         end if
      end if
    end subroutine quantum_numbers_undefine

    function quantum_numbers_undefined0 (qn, mask) result (qn_new)
      type(quantum_numbers_t), intent(in) :: qn
      type(quantum_numbers_mask_t), intent(in) :: mask
      type(quantum_numbers_t) :: qn_new
      qn_new = qn
      call quantum_numbers_undefine (qn_new, mask)
    end function quantum_numbers_undefined0

    function quantum_numbers_undefined1 (qn, mask) result (qn_new)
      type(quantum_numbers_t), dimension(:), intent(in) :: qn
      type(quantum_numbers_mask_t), intent(in) :: mask
      type(quantum_numbers_t), dimension(size(qn)) :: qn_new
      qn_new = qn
      call quantum_numbers_undefine (qn_new, mask)
    end function quantum_numbers_undefined1
```

```
function quantum_numbers_undefined11 (qn, mask) result (qn_new)
  type(quantum_numbers_t), dimension(:), intent(in) :: qn
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
  type(quantum_numbers_t), dimension(size(qn)) :: qn_new
  qn_new = qn
  call quantum_numbers_undefine (qn_new, mask)
end function quantum_numbers_undefined11
```

Return true if the input quantum number set has entries that would be removed by the applied mask, e.g., if polarization is defined but `mask%h` is set:

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_are_redundant
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_are_redundant (qn, mask) &
        result (redundant)
      logical :: redundant
      type(quantum_numbers_t), intent(in) :: qn
      type(quantum_numbers_mask_t), intent(in) :: mask
      redundant = .false.
      if (mask%f) then
        redundant = flavor_is_defined (qn%f)
      end if
      if (mask%c) then
        redundant = color_is_defined (qn%c)
      end if
      if (mask%h) then
        redundant = helicity_is_defined (qn%h)
      else if (mask%hd) then
        redundant = .not. helicity_is_diagonal (qn%h)
      end if
    end function quantum_numbers_are_redundant
```

Return true if the helicity flag is set or the diagonal-helicity flag is set.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_mask_diagonal_helicity
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_mask_diagonal_helicity (mask) &
        result (flag)
      logical :: flag
      type(quantum_numbers_mask_t), intent(in) :: mask
      flag = mask%h .or. mask%hd
    end function quantum_numbers_mask_diagonal_helicity
```

7.5 State matrices

This module deals with the internal state of a particle system, i.e., with its density matrix in flavor, color, and helicity space.

⟨`state_matrices.f90`⟩≡
  ⟨*File header*⟩

  ```
  module state_matrices
  ```

  ⟨*Use kinds*⟩
  ⟨*Use file utils*⟩
  ```
    use diagnostics !NODEP!
    use models
    use flavors
    use colors
    use helicities
    use quantum_numbers
  ```

  ⟨*Standard module head*⟩

  ⟨*State matrices: public*⟩

  ⟨*State matrices: parameters*⟩

  ⟨*State matrices: types*⟩

  ⟨*State matrices: interfaces*⟩

  ```
  contains
  ```

  ⟨*State matrices: procedures*⟩

  ```
  end module state_matrices
  ```

7.5.1 Nodes of the quantum state trie

A quantum state object represents an unnormalized density matrix, i.e., an array of possibilities for flavor, color, and helicity indices with associated complex values. Physically, the trace of this matrix is the summed squared matrix element for an interaction, and the matrix elements divided by this value correspond to the flavor-color-helicity density matrix. (Flavor and color are diagonal.)

We store density matrices as tries, that is, as trees where each branching represents the possible quantum numbers of a particle. The first branching is the first particle in the system. A leaf (the node corresponding to the last particle) contains the value of the matrix element.

Each node contains a flavor, color, and helicity entry. Note that each of those entries may be actually undefined, so we can also represent, e.g., unpolarized particles.

The value is meaningful only for leaves, which have no child nodes. There is a pointer to the parent node which allows for following the trie downwards from a leaf, it is null for a root node. The child nodes are implemented as a list,

so there is a pointer to the first and last child, and each node also has a `next` pointer to the next sibling.

The root node does not correspond to a particle, only its children do. The quantum numbers of the root node are irrelevant and will not be set. However, we use a common type for the three classes (root, branch, leaf); they may easily be distinguished by the association status of parent and child.

Node type

The node is linked in all directions: the parent, the first and last in the list of children, and the previous and next sibling. This allows us for adding and removing nodes and whole branches anywhere in the trie. (Circular links are not allowed, however.). The node holds its associated set of quantum numbers. The integer index, which is set only for leaf nodes, is the index of the corresponding matrix element value within the state matrix.

Temporarily, matrix-element values may be stored within a leaf node. This is used during state-matrix factorization. When the state matrix is `freeze`d, these values are transferred to the matrix-element array within the host state matrix.

⟨*State matrices: types*⟩≡

```
type :: node_t
   private
   type(quantum_numbers_t) :: qn
   type(node_t), pointer :: parent => null ()
   type(node_t), pointer :: child_first => null ()
   type(node_t), pointer :: child_last => null ()
   type(node_t), pointer :: next => null ()
   type(node_t), pointer :: previous => null ()
   integer :: me_index = 0
   integer, dimension(:), allocatable :: me_count
   complex(default) :: me = 0
end type node_t
```

Operations on nodes

Recursively deallocate all children of the current node. This includes any values associated with the children.

⟨*State matrices: procedures*⟩≡

```
pure recursive subroutine node_delete_offspring (node)
   type(node_t), pointer :: node
   type(node_t), pointer :: child
   child => node%child_first
   do while (associated (child))
      node%child_first => node%child_first%next
      call node_delete_offspring (child)
      deallocate (child)
   end do
   node%child_last => null ()
end subroutine node_delete_offspring
```

Remove a node including its offspring. Adjust the pointers of parent and siblings, if necessary.

⟨*State matrices: procedures*⟩+≡

```
pure subroutine node_delete (node)
  type(node_t), pointer :: node
  call node_delete_offspring (node)
  if (associated (node%previous)) then
     node%previous%next => node%next
  else if (associated (node%parent)) then
     node%parent%child_first => node%next
  end if
  if (associated (node%next)) then
     node%next%previous => node%previous
  else if (associated (node%parent)) then
     node%parent%child_last => node%previous
  end if
  deallocate (node)
end subroutine node_delete
```

Append a child node

⟨*State matrices: procedures*⟩+≡

```
subroutine node_append_child (node, child)
  type(node_t), target, intent(inout) :: node
  type(node_t), pointer :: child
  allocate (child)
  if (associated (node%child_last)) then
     node%child_last%next => child
     child%previous => node%child_last
  else
     node%child_first => child
  end if
  node%child_last => child
  child%parent => node
end subroutine node_append_child
```

I/O

Output of a single node, no recursion. We print the quantum numbers in square brackets, then the value (if any).

⟨*State matrices: procedures*⟩+≡

```
subroutine node_write (node, me_array, verbose, unit)
  type(node_t), intent(in) :: node
  complex(default), dimension(:), intent(in), optional :: me_array
  logical, intent(in), optional :: verbose
  integer, intent(in), optional :: unit
  logical :: verb
  integer :: u
  verb = .false.;  if (present (verbose)) verb = verbose
  u = output_unit (unit);  if (u < 0)  return
  call quantum_numbers_write (node%qn, u)
  if (node%me_index /= 0) then
     write (u, "(A,I0,A)", advance="no")  " => ME(", node%me_index, ")"
```

```fortran
      if (present (me_array)) then
         write (u, "(A)", advance="no")  " = "
         if (aimag (me_array(node%me_index)) == 0) then
            write (u, "(1P,G19.12)", advance="no")  real (me_array(node%me_index))
         else
            write (u, "(1P,G19.12,',',G19.12)", advance="no") me_array(node%me_index)
         end if
      end if
   end if
   write (u, *)
   if (verb) then
      call ptr_write ("parent      ", node%parent)
      call ptr_write ("child_first", node%child_first)
      call ptr_write ("child_last ", node%child_last)
      call ptr_write ("next       ", node%next)
      call ptr_write ("previous   ", node%previous)
   end if
 contains
   subroutine ptr_write (label, node)
     character(*), intent(in) :: label
     type(node_t), pointer :: node
     if (associated (node)) then
        write (u, "(10x,A,1x,'->',1x)", advance="no") label
        call quantum_numbers_write (node%qn, u)
        write (u, *)
     end if
   end subroutine ptr_write
 end subroutine node_write
```

Recursive output of a node:

⟨*State matrices: procedures*⟩+≡

```fortran
   recursive subroutine node_write_rec (node, me_array, verbose, indent, unit)
     type(node_t), intent(in), target :: node
     complex(default), dimension(:), intent(in), optional :: me_array
     logical, intent(in), optional :: verbose
     integer, intent(in), optional :: indent
     integer, intent(in), optional :: unit
     type(node_t), pointer :: current
     logical :: verb
     integer :: i, u
     verb = .false.;  if (present (verbose))  verb = verbose
     i = 0;  if (present (indent)) i = indent
     u = output_unit (unit);  if (u < 0)  return
     current => node%child_first
     do while (associated (current))
        write (u, "(A)", advance="no")  repeat (" ", i)
        call node_write (current, me_array, verb, u)
        call node_write_rec (current, me_array, verb, i+2, u)
        current => current%next
     end do
   end subroutine node_write_rec
```

Binary I/O. Matrix elements are written only for leaf nodes. this was actually

a pointer is lost.

⟨*State matrices: procedures*⟩+≡

```fortran
recursive subroutine node_write_raw_rec (node, u)
  type(node_t), intent(in), target :: node
  integer, intent(in) :: u
  logical :: associated_child_first, associated_next
  call quantum_numbers_write_raw (node%qn, u)
  associated_child_first = associated (node%child_first)
  write (u) associated_child_first
  associated_next = associated (node%next)
  write (u) associated_next
  if (associated_child_first) then
     call node_write_raw_rec (node%child_first, u)
  else
     write (u)  node%me_index
  end if
  if (associated_next) then
     call node_write_raw_rec (node%next, u)
  end if
end subroutine node_write_raw_rec

recursive subroutine node_read_raw_rec (node, u, parent, iostat)
  type(node_t), intent(out), target :: node
  integer, intent(in) :: u
  type(node_t), intent(in), optional, target :: parent
  integer, intent(out), optional :: iostat
  logical :: associated_child_first, associated_next
  type(node_t), pointer :: child
  call quantum_numbers_read_raw (node%qn, u, iostat=iostat)
  read (u, iostat=iostat) associated_child_first
  read (u, iostat=iostat) associated_next
  if (present (parent))  node%parent => parent
  if (associated_child_first) then
     allocate (child)
     node%child_first => child
     node%child_last => null ()
     call node_read_raw_rec (child, u, node, iostat=iostat)
     do while (associated (child))
        child%previous => node%child_last
        node%child_last => child
        child => child%next
     end do
  else
     read (u, iostat=iostat)  node%me_index
  end if
  if (associated_next) then
     allocate (node%next)
     call node_read_raw_rec (node%next, u, parent, iostat=iostat)
  end if
end subroutine node_read_raw_rec
```

7.5.2 State matrix

Definition

The quantum state object is a container that keeps and hides the root node. For direct accessibility of values, they are stored in a separate array. The leaf nodes of the quantum-number tree point to those values, once the state matrix is finalized.

The `norm` component is redefined if a common factor is extracted from all nodes.

⟨*State matrices: public*⟩≡
```
    public :: state_matrix_t
```

⟨*State matrices: types*⟩+≡
```
    type :: state_matrix_t
      private
      type(node_t), pointer :: root => null ()
      integer :: depth = 0
      integer :: n_matrix_elements = 0
      logical :: leaf_nodes_store_values = .false.
      integer :: n_counters = 0
      complex(default), dimension(:), allocatable :: me
      real(default) :: norm = 1
    end type state_matrix_t
```

This initializer allocates the root node but does not fill anything. We declare whether values are stored within the nodes during state-matrix construction, and how many counters should be maintained (default: none).

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_init
```

⟨*State matrices: procedures*⟩+≡
```
    elemental subroutine state_matrix_init (state, store_values, n_counters)
      type(state_matrix_t), intent(out) :: state
      logical, intent(in), optional :: store_values
      integer, intent(in), optional :: n_counters
      allocate (state%root)
      if (present (store_values)) &
         state%leaf_nodes_store_values = store_values
      if (present (n_counters)) state%n_counters = n_counters
    end subroutine state_matrix_init
```

This recursively deletes all children of the root node, restoring the initial state. The matrix element array is not finalized, since it does not contain physical entries, just pointers.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_final
```

⟨*State matrices: procedures*⟩+≡
```
    elemental subroutine state_matrix_final (state)
      type(state_matrix_t), intent(inout) :: state
      if (allocated (state%me))  deallocate (state%me)
      if (associated (state%root))  call node_delete (state%root)
      state%depth = 0
```

```
      state%n_matrix_elements = 0
    end subroutine state_matrix_final
```

Output: Present the tree as a nested list with appropriate indentation.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_write
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_write (state, unit, write_value_list, verbose)
      type(state_matrix_t), intent(in) :: state
      logical, intent(in), optional :: write_value_list, verbose
      integer, intent(in), optional :: unit
      integer :: u
      integer :: i
      u = output_unit (unit);  if (u < 0)  return
      write (u, *) "State matrix:  norm = ", state%norm
      if (associated (state%root)) then
         if (allocated (state%me)) then
            call node_write_rec (state%root, state%me, verbose, 1, u)
         else
            call node_write_rec (state%root, verbose=verbose, indent=1, unit=u)
         end if
      end if
      if (present (write_value_list)) then
         if (write_value_list .and. allocated (state%me)) then
            do i = 1, size (state%me)
               write (u, "(1x,I0,A)", advance="no")  i, ":"
               write (u, *)  state%me(i)
            end do
         end if
      end if
    end subroutine state_matrix_write
```

Binary I/O. The auxiliary matrix-element array is not written, but reconstructed after reading the tree.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_write_raw
    public :: state_matrix_read_raw
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_write_raw (state, u)
      type(state_matrix_t), intent(in) :: state
      integer, intent(in) :: u
      logical :: associated_root
      associated_root = associated (state%root)
      write (u) associated_root
      if (associated_root) then
         write (u) state%depth
         write (u) state%norm
         call node_write_raw_rec (state%root, u)
      end if
    end subroutine state_matrix_write_raw

    subroutine state_matrix_read_raw (state, u, iostat)
```

```
      type(state_matrix_t), intent(out) :: state
      integer, intent(in) :: u
      integer, intent(out), optional :: iostat
      logical :: associated_root
      read (u, iostat=iostat) associated_root
      if (associated_root) then
         read (u, iostat=iostat) state%depth
         read (u, iostat=iostat) state%norm
         call state_matrix_init (state)
         call node_read_raw_rec (state%root, u, iostat=iostat)
         call state_matrix_freeze (state)
      end if
   end subroutine state_matrix_read_raw
```

Properties of the quantum state

A state is defined if its root is allocated:

⟨*State matrices: public*⟩+≡
```
   public :: state_matrix_is_defined
```

⟨*State matrices: procedures*⟩+≡
```
   elemental function state_matrix_is_defined (state) result (defined)
     logical :: defined
     type(state_matrix_t), intent(in) :: state
     defined = associated (state%root)
   end function state_matrix_is_defined
```

A state is empty if its depth is zero:

⟨*State matrices: public*⟩+≡
```
   public :: state_matrix_is_empty
```

⟨*State matrices: procedures*⟩+≡
```
   elemental function state_matrix_is_empty (state) result (flag)
     logical :: flag
     type(state_matrix_t), intent(in) :: state
     flag = state%depth == 0
   end function state_matrix_is_empty
```

Return the number of matrix-element values.

⟨*State matrices: public*⟩+≡
```
   public :: state_matrix_get_n_matrix_elements
```

⟨*State matrices: procedures*⟩+≡
```
   function state_matrix_get_n_matrix_elements (state) result (n)
     integer :: n
     type(state_matrix_t), intent(in) :: state
     n = state%n_matrix_elements
   end function state_matrix_get_n_matrix_elements
```

Return the number of leaves. This can be larger than the number of independent matrix elements.

⟨*State matrices: public*⟩+≡
```
   public :: state_matrix_get_n_leaves
```

⟨*State matrices: procedures*⟩+≡
```
    function state_matrix_get_n_leaves (state) result (n)
      integer :: n
      type(state_matrix_t), intent(in) :: state
      type(state_iterator_t) :: it
      n = 0
      call state_iterator_init (it, state)
      do while (state_iterator_is_valid (it))
         n = n + 1
         call state_iterator_advance (it)
      end do
    end function state_matrix_get_n_leaves
```

Return the depth:

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_get_depth
```

⟨*State matrices: procedures*⟩+≡
```
    function state_matrix_get_depth (state) result (depth)
      integer :: depth
      type(state_matrix_t), intent(in) :: state
      depth = state%depth
    end function state_matrix_get_depth
```

Return the norm:

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_get_norm
```

⟨*State matrices: procedures*⟩+≡
```
    function state_matrix_get_norm (state) result (norm)
      real(default) :: norm
      type(state_matrix_t), intent(in) :: state
      norm = state%norm
    end function state_matrix_get_norm
```

Retrieving contents

Return the quantum number array, using an index. We have to scan the state matrix since there is no shortcut.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_get_quantum_numbers
```

⟨*State matrices: procedures*⟩+≡
```
    function state_matrix_get_quantum_numbers (state, i) result (qn)
      type(state_matrix_t), intent(in), target :: state
      integer, intent(in) :: i
      type(quantum_numbers_t), dimension(state%depth) :: qn
      type(state_iterator_t) :: it
      integer :: k
      k = 0
      call state_iterator_init (it, state)
      do while (state_iterator_is_valid (it))
```

```
        k = k + 1
        if (k == i) then
           qn = state_iterator_get_quantum_numbers (it)
           return
        end if
        call state_iterator_advance (it)
     end do
   end function state_matrix_get_quantum_numbers
```

Return a single matrix element using its index. Works only if the shortcut array is allocated.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_get_matrix_element
```

⟨*State matrices: procedures*⟩+≡
```
    function state_matrix_get_matrix_element (state, i) result (me)
      complex(default) :: me
      type(state_matrix_t), intent(in) :: state
      integer, intent(in) :: i
      if (allocated (state%me)) then
         me = state%me(i)
      else
         me = 0
      end if
    end function state_matrix_get_matrix_element
```

Return the color index with maximum absolute value that is present within the state matrix.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_get_max_color_value
```

⟨*State matrices: procedures*⟩+≡
```
    function state_matrix_get_max_color_value (state) result (cmax)
      integer :: cmax
      type(state_matrix_t), intent(in) :: state
      if (associated (state%root)) then
         cmax = node_get_max_color_value (state%root)
      else
         cmax = 0
      end if
    contains
      recursive function node_get_max_color_value (node) result (cmax)
        integer :: cmax
        type(node_t), intent(in), target :: node
        type(node_t), pointer :: current
        cmax = quantum_numbers_get_max_color_value (node%qn)
        current => node%child_first
        do while (associated (current))
           cmax = max (cmax, node_get_max_color_value (current))
           current => current%next
        end do
      end function node_get_max_color_value
    end function state_matrix_get_max_color_value
```

Building the quantum state

The procedure generates a branch associated to the input array of quantum numbers. If the branch exists already, it is used.

Optionally, we set the matrix-element index, a value (which may be added to the previous one), and increment one of the possible counters. We may also return the matrix element index of the current node.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_add_state
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_add_state &
        (state, qn, index, value, sum_values, counter_index, me_index)
      type(state_matrix_t), intent(inout) :: state
      type(quantum_numbers_t), dimension(:), intent(in) :: qn
      integer, intent(in), optional :: index
      complex(default), intent(in), optional :: value
      logical, intent(in), optional :: sum_values
      integer, intent(in), optional :: counter_index
      integer, intent(out), optional :: me_index
      logical :: set_index, get_index, add
      set_index = present (index)
      get_index = present (me_index)
      add = .false.;  if (present (sum_values))  add = sum_values
      if (state%depth == 0) then
         state%depth = size (qn)
      else if (state%depth /= size (qn)) then
         call state_matrix_write (state)
         call msg_bug ("State matrix: depth mismatch")
      end if
      if (size (qn) > 0)  call node_make_branch (state%root, qn)
    contains
      recursive subroutine node_make_branch (parent, qn)
        type(node_t), pointer :: parent
        type(quantum_numbers_t), dimension(:), intent(in) :: qn
        type(node_t), pointer :: child
        logical :: match
        match = .false.
        child => parent%child_first
        SCAN_CHILDREN: do while (associated (child))
           match = child%qn == qn(1)
           if (match)  exit SCAN_CHILDREN
           child => child%next
        end do SCAN_CHILDREN
        if (.not. match) then
           call node_append_child (parent, child)
           child%qn = qn(1)
        end if
        select case (size (qn))
        case (1)
           if (.not. match) then
              state%n_matrix_elements = state%n_matrix_elements + 1
              child%me_index = state%n_matrix_elements
           end if
```

```
            if (set_index) then
                child%me_index = index
            end if
            if (get_index) then
                me_index = child%me_index
            end if
            if (present (counter_index)) then
                if (.not. allocated (child%me_count)) then
                    allocate (child%me_count (state%n_counters))
                    child%me_count = 0
                end if
                child%me_count(counter_index) = child%me_count(counter_index) + 1
            end if
            if (present (value)) then
                if (add) then
                    child%me = child%me + value
                else
                    child%me = value
                end if
            end if
         case (2:)
            call node_make_branch (child, qn(2:))
         end select
       end subroutine node_make_branch
    end subroutine state_matrix_add_state
```

Remove irrelevant flavor/color/helicity labels and the corresponding branchings. The masks indicate which particles are affected; the masks length should coincide with the depth of the trie (without the root node). Recursively scan the whole tree, starting from the leaf nodes and working up to the root node. If a mask entry is set for the current tree level, scan the children there. For each child within that level make a new empty branch where the masked quantum number is undefined. Then recursively combine all following children with matching quantum number into this new node and move on.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_collapse
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_collapse (state, mask)
      type(state_matrix_t), intent(inout) :: state
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
      type(state_matrix_t) :: red_state
      if (state_matrix_is_defined (state)) then
         call state_matrix_reduce (state, mask, red_state)
         call state_matrix_final (state)
         state = red_state
      end if
    end subroutine state_matrix_collapse
```

Transform the given state matrix into a reduced state matrix where some quantum numbers are removed, as indicated by the mask. The procedure creates a new state matrix, so the old one can be deleted after this if it is no longer used.

⟨*State matrices: public*⟩+≡

```
      public :: state_matrix_reduce
```
⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_reduce (state, mask, red_state)
      type(state_matrix_t), intent(in), target :: state
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
      type(state_matrix_t), intent(out) :: red_state
      type(state_iterator_t) :: it
      type(quantum_numbers_t), dimension(size(mask)) :: qn
      call state_matrix_init (red_state)
      call state_iterator_init (it, state)
      do while (state_iterator_is_valid (it))
         qn = state_iterator_get_quantum_numbers (it)
         call quantum_numbers_undefine (qn, mask)
         call state_matrix_add_state (red_state, qn)
         call state_iterator_advance (it)
      end do
    end subroutine state_matrix_reduce
```

This subroutine sets up the matrix-element array. The leaf nodes aquire the index values that point to the appropriate matrix-element entry.

We recursively scan the trie. Once we arrive at a leaf node, the index is increased and associated to that node. Finally, we allocate the matrix-element array with the appropriate size.

If matrix element values are temporarily stored within the leaf nodes, we scan the state again and transfer them to the matrix-element array.

⟨*State matrices: public*⟩+≡
```
      public :: state_matrix_freeze
```
⟨*State matrices: interfaces*⟩≡
```
    interface state_matrix_freeze
       module procedure state_matrix_freeze1
       module procedure state_matrix_freeze2
    end interface
```
⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_freeze1 (state)
      type(state_matrix_t), intent(inout), target :: state
      type(state_iterator_t) :: it
      if (associated (state%root)) then
         if (allocated (state%me))  deallocate (state%me)
         allocate (state%me (state%n_matrix_elements))
         state%me = 0
      end if
      if (state%leaf_nodes_store_values) then
         call state_iterator_init (it, state)
         do while (state_iterator_is_valid (it))
            state%me(state_iterator_get_me_index (it)) &
                 = state_iterator_get_matrix_element (it)
            call state_iterator_advance (it)
         end do
         state%leaf_nodes_store_values = .false.
      end if
    end subroutine state_matrix_freeze1
```

```
subroutine state_matrix_freeze2 (state)
  type(state_matrix_t), dimension(:), intent(inout), target :: state
  integer :: i
  do i = 1, size (state)
     call state_matrix_freeze1 (state(i))
  end do
end subroutine state_matrix_freeze2
```

Direct access to the value array

Several methods for setting a value directly are summarized in this generic:

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_set_matrix_element
```

⟨*State matrices: interfaces*⟩+≡
```
    interface state_matrix_set_matrix_element
       module procedure state_matrix_set_matrix_element_qn
       module procedure state_matrix_set_matrix_element_all
       module procedure state_matrix_set_matrix_element_array
       module procedure state_matrix_set_matrix_element_single
    end interface
```

Set a value that corresponds to a quantum number array:

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_set_matrix_element_qn (state, qn, value)
      type(state_matrix_t), intent(inout), target :: state
      type(quantum_numbers_t), dimension(:), intent(in) :: qn
      complex(default), intent(in) :: value
      type(state_iterator_t) :: it
      call state_iterator_init (it, state)
      do while (state_iterator_is_valid (it))
         if (all (qn == state_iterator_get_quantum_numbers (it))) then
            call state_iterator_set_matrix_element (it, value)
            return
         end if
         call state_iterator_advance (it)
      end do
    end subroutine state_matrix_set_matrix_element_qn
```

Set all matrix elements to a single value

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_set_matrix_element_all (state, value)
      type(state_matrix_t), intent(inout) :: state
      complex(default), intent(in) :: value
      state%me = value
    end subroutine state_matrix_set_matrix_element_all
```

Set the matrix-element array directly.

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_set_matrix_element_array (state, value)
      type(state_matrix_t), intent(inout) :: state
      complex(default), dimension(:), intent(in) :: value
```

```
      state%me = value
    end subroutine state_matrix_set_matrix_element_array

    pure subroutine state_matrix_set_matrix_element_single (state, i, value)
      type(state_matrix_t), intent(inout) :: state
      integer, intent(in) :: i
      complex(default), intent(in) :: value
      state%me(i) = value
    end subroutine state_matrix_set_matrix_element_single
```

Add a value to a matrix element

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_add_to_matrix_element
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_add_to_matrix_element (state, i, value)
      type(state_matrix_t), intent(inout) :: state
      integer, intent(in) :: i
      complex(default), intent(in) :: value
      state%me(i) = state%me(i) + value
    end subroutine state_matrix_add_to_matrix_element
```

7.5.3   State iterators

Accessing the quantum state from outside is best done using a specialized itera-
tor, i.e., a pointer to a particular branch of the quantum state trie. Technically,
the iterator contains a pointer to a leaf node, but via parent pointers it allows
to access the whole branch where the leaf is attached. For quick access, we also
keep the branch depth (which is assumed to be universal for a quantum state).

⟨*State matrices: public*⟩+≡
```
    public :: state_iterator_t
```

⟨*State matrices: types*⟩+≡
```
    type :: state_iterator_t
      private
      integer :: depth = 0
      type(state_matrix_t), pointer :: state => null ()
      type(node_t), pointer :: node => null ()
    end type state_iterator_t
```

The initializer: Point at the first branch. Note that this cannot be pure, thus
not be elemental, because the iterator can be used to manipulate data in the
state matrix.

⟨*State matrices: public*⟩+≡
```
    public :: state_iterator_init
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_iterator_init (it, state)
      type(state_iterator_t), intent(out) :: it
      type(state_matrix_t), intent(in), target :: state
      it%state => state
      it%depth = state%depth
```

```
         if (state_matrix_is_defined (state)) then
            it%node => state%root
            do while (associated (it%node%child_first))
               it%node => it%node%child_first
            end do
         else
            it%node => null ()
         end if
      end subroutine state_iterator_init
```

Go forward. Recursively programmed: if the next node does not exist, go back
to the parent node and look at its successor (if present), etc.

There is a possible pitfall in the implementation: If the dummy pointer
argument to the `find_next` routine is used directly, we still get the correct
result for the iterator, but calling the recursion on `node%parent` means that
we manipulate a parent pointer in the original state in addition to the iterator.
Making a local copy of the pointer avoids this. Using pointer intent would be
helpful, but we do not yet rely on this F2003 feature.

⟨*State matrices: public*⟩+≡
```
      public :: state_iterator_advance
```

⟨*State matrices: procedures*⟩+≡
```
      subroutine state_iterator_advance (it)
        type(state_iterator_t), intent(inout) :: it
        call find_next (it%node)
      contains
        recursive subroutine find_next (node_in)
          type(node_t), intent(in), target :: node_in
          type(node_t), pointer :: node
          node => node_in
          if (associated (node%next)) then
             node => node%next
             do while (associated (node%child_first))
                node => node%child_first
             end do
             it%node => node
          else if (associated (node%parent)) then
             call find_next (node%parent)
          else
             it%node => null ()
          end if
        end subroutine find_next
      end subroutine state_iterator_advance
```

If all has been scanned, the iterator is at an undefined state. Check for this:

⟨*State matrices: public*⟩+≡
```
      public :: state_iterator_is_valid
```

⟨*State matrices: procedures*⟩+≡
```
      function state_iterator_is_valid (it) result (defined)
        logical :: defined
        type(state_iterator_t), intent(in) :: it
        defined = associated (it%node)
```

```
      end function state_iterator_is_valid
```

Return the matrix-element index that corresponds to the current node

⟨*State matrices: public*⟩+≡
```
    public :: state_iterator_get_me_index
```

⟨*State matrices: procedures*⟩+≡
```
    function state_iterator_get_me_index (it) result (n)
      integer :: n
      type(state_iterator_t), intent(in) :: it
      n = it%node%me_index
    end function state_iterator_get_me_index
```

Return the number of times this quantum-number state has been added (noting that it is physically inserted only the first time). Note that for each state, there is an array of counters.

⟨*State matrices: public*⟩+≡
```
    public :: state_iterator_get_me_count
```

⟨*State matrices: procedures*⟩+≡
```
    function state_iterator_get_me_count (it) result (n)
      integer, dimension(:), allocatable :: n
      type(state_iterator_t), intent(in) :: it
      if (allocated (it%node%me_count)) then
         allocate (n (size (it%node%me_count)))
         n = it%node%me_count
      else
         allocate (n (0))
      end if
    end function state_iterator_get_me_count
```

Use the iterator to retrieve quantum-number information:

⟨*State matrices: public*⟩+≡
```
    public :: state_iterator_get_quantum_numbers
    public :: state_iterator_get_flavor
    public :: state_iterator_get_color
    public :: state_iterator_get_helicity
```

⟨*State matrices: interfaces*⟩+≡
```
    interface state_iterator_get_quantum_numbers
       module procedure state_iterator_get_qn_multi
       module procedure state_iterator_get_qn_slice
       module procedure state_iterator_get_qn_range
       module procedure state_iterator_get_qn_single
    end interface

    interface state_iterator_get_flavor
       module procedure state_iterator_get_flv_multi
       module procedure state_iterator_get_flv_slice
       module procedure state_iterator_get_flv_range
       module procedure state_iterator_get_flv_single
    end interface
```

```
    interface state_iterator_get_color
       module procedure state_iterator_get_col_multi
       module procedure state_iterator_get_col_slice
       module procedure state_iterator_get_col_range
       module procedure state_iterator_get_col_single
    end interface

    interface state_iterator_get_helicity
       module procedure state_iterator_get_hel_multi
       module procedure state_iterator_get_hel_slice
       module procedure state_iterator_get_hel_range
       module procedure state_iterator_get_hel_single
    end interface
```

These versions return the whole quantum number array

⟨*State matrices: procedures*⟩+≡

```
    function state_iterator_get_qn_multi (it) result (qn)
      type(state_iterator_t), intent(in) :: it
      type(quantum_numbers_t), dimension(it%depth) :: qn
      type(node_t), pointer :: node
      integer :: i
      node => it%node
      do i = it%depth, 1, -1
         qn(i) = node%qn
         node => node%parent
      end do
    end function state_iterator_get_qn_multi

    function state_iterator_get_flv_multi (it) result (flv)
      type(state_iterator_t), intent(in) :: it
      type(flavor_t), dimension(it%depth) :: flv
      flv = quantum_numbers_get_flavor &
           (state_iterator_get_quantum_numbers (it))
    end function state_iterator_get_flv_multi

    function state_iterator_get_col_multi (it) result (col)
      type(state_iterator_t), intent(in) :: it
      type(color_t), dimension(it%depth) :: col
      col = quantum_numbers_get_color &
           (state_iterator_get_quantum_numbers (it))
    end function state_iterator_get_col_multi

    function state_iterator_get_hel_multi (it) result (hel)
      type(state_iterator_t), intent(in) :: it
      type(helicity_t), dimension(it%depth) :: hel
      hel = quantum_numbers_get_helicity &
           (state_iterator_get_quantum_numbers (it))
    end function state_iterator_get_hel_multi
```

An array slice (derived from the above).

⟨*State matrices: procedures*⟩+≡

```
    function state_iterator_get_qn_slice (it, index) result (qn)
      type(state_iterator_t), intent(in) :: it
```

```
    integer, dimension(:), intent(in) :: index
    type(quantum_numbers_t), dimension(size(index)) :: qn
    type(quantum_numbers_t), dimension(it%depth) :: qn_tmp
    qn_tmp = state_iterator_get_qn_multi (it)
    qn = qn_tmp(index)
  end function state_iterator_get_qn_slice

  function state_iterator_get_flv_slice (it, index) result (flv)
    type(state_iterator_t), intent(in) :: it
    integer, dimension(:), intent(in) :: index
    type(flavor_t), dimension(size(index)) :: flv
    flv = quantum_numbers_get_flavor &
         (state_iterator_get_quantum_numbers (it, index))
  end function state_iterator_get_flv_slice

  function state_iterator_get_col_slice (it, index) result (col)
    type(state_iterator_t), intent(in) :: it
    integer, dimension(:), intent(in) :: index
    type(color_t), dimension(size(index)) :: col
    col = quantum_numbers_get_color &
         (state_iterator_get_quantum_numbers (it, index))
  end function state_iterator_get_col_slice

  function state_iterator_get_hel_slice (it, index) result (hel)
    type(state_iterator_t), intent(in) :: it
    integer, dimension(:), intent(in) :: index
    type(helicity_t), dimension(size(index)) :: hel
    hel = quantum_numbers_get_helicity &
         (state_iterator_get_quantum_numbers (it, index))
  end function state_iterator_get_hel_slice
```

An array range (implemented directly).

⟨*State matrices: procedures*⟩+≡

```
    function state_iterator_get_qn_range (it, k1, k2) result (qn)
      type(state_iterator_t), intent(in) :: it
      integer, intent(in) :: k1, k2
      type(quantum_numbers_t), dimension(k2-k1+1) :: qn
      type(node_t), pointer :: node
      integer :: i
      node => it%node
      SCAN: do i = it%depth, 1, -1
         if (k1 <= i .and. i <= k2) then
            qn(i-k1+1) = node%qn
         else
            node => node%parent
         end if
      end do SCAN
    end function state_iterator_get_qn_range

    function state_iterator_get_flv_range (it, k1, k2) result (flv)
      type(state_iterator_t), intent(in) :: it
      integer, intent(in) :: k1, k2
      type(flavor_t), dimension(k2-k1+1) :: flv
      flv = quantum_numbers_get_flavor &
```

```
                (state_iterator_get_quantum_numbers (it, k1, k2))
      end function state_iterator_get_flv_range

      function state_iterator_get_col_range (it, k1, k2) result (col)
        type(state_iterator_t), intent(in) :: it
        integer, intent(in) :: k1, k2
        type(color_t), dimension(k2-k1+1) :: col
        col = quantum_numbers_get_color &
              (state_iterator_get_quantum_numbers (it, k1, k2))
      end function state_iterator_get_col_range

      function state_iterator_get_hel_range (it, k1, k2) result (hel)
        type(state_iterator_t), intent(in) :: it
        integer, intent(in) :: k1, k2
        type(helicity_t), dimension(k2-k1+1) :: hel
        hel = quantum_numbers_get_helicity &
              (state_iterator_get_quantum_numbers (it, k1, k2))
      end function state_iterator_get_hel_range
```

Just a specific single element

⟨*State matrices: procedures*⟩+≡

```
      function state_iterator_get_qn_single (it, k) result (qn)
        type(state_iterator_t), intent(in) :: it
        integer, intent(in) :: k
        type(quantum_numbers_t) :: qn
        type(node_t), pointer :: node
        integer :: i
        node => it%node
        SCAN: do i = it%depth, 1, -1
           if (i == k) then
              qn = node%qn
              exit SCAN
           else
              node => node%parent
           end if
        end do SCAN
      end function state_iterator_get_qn_single

      function state_iterator_get_flv_single (it, k) result (flv)
        type(state_iterator_t), intent(in) :: it
        integer, intent(in) :: k
        type(flavor_t) :: flv
        flv = quantum_numbers_get_flavor &
              (state_iterator_get_quantum_numbers (it, k))
      end function state_iterator_get_flv_single

      function state_iterator_get_col_single (it, k) result (col)
        type(state_iterator_t), intent(in) :: it
        integer, intent(in) :: k
        type(color_t) :: col
        col = quantum_numbers_get_color &
              (state_iterator_get_quantum_numbers (it, k))
      end function state_iterator_get_col_single
```

```
function state_iterator_get_hel_single (it, k) result (hel)
  type(state_iterator_t), intent(in) :: it
  integer, intent(in) :: k
  type(helicity_t) :: hel
  hel = quantum_numbers_get_helicity &
       (state_iterator_get_quantum_numbers (it, k))
end function state_iterator_get_hel_single
```

Retrieve the matrix element value associated with the current node.

⟨*State matrices: public*⟩+≡
```
    public :: state_iterator_get_matrix_element
```

⟨*State matrices: procedures*⟩+≡
```
    function state_iterator_get_matrix_element (it) result (me)
      complex(default) :: me
      type(state_iterator_t), intent(in) :: it
      if (it%state%leaf_nodes_store_values) then
         me = it%node%me
      else if (it%node%me_index /= 0) then
         me = it%state%me(it%node%me_index)
      else
         me = 0
      end if
    end function state_iterator_get_matrix_element
```

Set the matrix element value using the state iterator.

⟨*State matrices: public*⟩+≡
```
    public :: state_iterator_set_matrix_element
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_iterator_set_matrix_element (it, value)
      type(state_iterator_t), intent(inout) :: it
      complex(default), intent(in) :: value
      if (it%node%me_index /= 0) then
         it%state%me(it%node%me_index) = value
      end if
    end subroutine state_iterator_set_matrix_element
```

7.5.4   Operations on quantum states

Return a deep copy of a state matrix.

⟨*State matrices: public*⟩+≡
```
    public :: assignment(=)
```

⟨*State matrices: interfaces*⟩+≡
```
    interface assignment(=)
       module procedure state_matrix_assign
    end interface
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_assign (state_out, state_in)
  type(state_matrix_t), intent(out) :: state_out
  type(state_matrix_t), intent(in), target :: state_in
  type(state_iterator_t) :: it
  if (.not. state_matrix_is_defined (state_in))  return
  call state_matrix_init (state_out)
  call state_iterator_init (it, state_in)
  do while (state_iterator_is_valid (it))
     call state_matrix_add_state (state_out, &
           state_iterator_get_quantum_numbers (it), &
           state_iterator_get_me_index (it))
     call state_iterator_advance (it)
  end do
  if (allocated (state_in%me)) then
     allocate (state_out%me (size (state_in%me)))
     state_out%me = state_in%me
  end if
end subroutine state_matrix_assign
```

Normalize all matrix elements, i.e., multiply by a common factor. Assuming that the factor is nonzero, of course.

⟨*State matrices: public*⟩+≡

```
public :: state_matrix_renormalize
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_renormalize (state, factor)
  type(state_matrix_t), intent(inout) :: state
  complex(default), intent(in) :: factor
  state%me = state%me * factor
end subroutine state_matrix_renormalize
```

Renormalize the state matrix by its trace, if nonzero. The renormalization is reflected in the state-matrix norm.

⟨*State matrices: public*⟩+≡

```
public :: state_matrix_normalize_by_trace
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_normalize_by_trace (state)
  type(state_matrix_t), intent(inout) :: state
  real(default) :: trace
  trace = state_matrix_trace (state)
  if (trace /= 0) then
     state%me = state%me / trace
     state%norm = state%norm * trace
  end if
end subroutine state_matrix_normalize_by_trace
```

Analogous, but renormalize by maximal (absolute) value.

⟨*State matrices: public*⟩+≡

```
public :: state_matrix_normalize_by_max
```

584

⟨*State matrices: procedures*⟩+≡
```
subroutine state_matrix_normalize_by_max (state)
  type(state_matrix_t), intent(inout) :: state
  real(default) :: m
  m = maxval (abs (state%me))
  if (m /= 0) then
     state%me = state%me / m
     state%norm = state%norm * m
  end if
end subroutine state_matrix_normalize_by_max
```

Return the sum of all matrix element values.

⟨*State matrices: public*⟩+≡
```
public :: state_matrix_sum
```

⟨*State matrices: procedures*⟩+≡
```
function state_matrix_sum (state) result (value)
  complex(default) :: value
  type(state_matrix_t), intent(in) :: state
  value = sum (state%me)
end function state_matrix_sum
```

Return the trace of a state matrix, i.e., the sum over all diagonal values. If `qn_in` is provided, only branches that match this quantum-numbers array are considered.

⟨*State matrices: public*⟩+≡
```
public :: state_matrix_trace
```

⟨*State matrices: procedures*⟩+≡
```
function state_matrix_trace (state, qn_in) result (trace)
  complex(default) :: trace
  type(state_matrix_t), intent(in), target :: state
  type(quantum_numbers_t), dimension(:), intent(in), optional :: qn_in
  type(quantum_numbers_t), dimension(:), allocatable :: qn
  type(state_iterator_t) :: it
  allocate (qn (state_matrix_get_depth (state)))
  trace = 0
  call state_iterator_init (it, state)
  do while (state_iterator_is_valid (it))
     qn = state_iterator_get_quantum_numbers (it)
     if (present (qn_in)) then
        if (.not. all (qn .match. qn_in)) then
           call state_iterator_advance (it);  cycle
        end if
     end if
     if (all (quantum_numbers_are_diagonal (qn))) then
        trace = trace + state_iterator_get_matrix_element (it)
     end if
     call state_iterator_advance (it)
  end do
end function state_matrix_trace
```

Append new states which are color-contracted versions of the existing states. The matrix element index of each color contraction coincides with the index of its origin, so no new matrix elements are generated. After this operation, no `freeze` must be performed anymore.

⟨*State matrices: public*⟩+≡

```
public :: state_matrix_add_color_contractions
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_add_color_contractions (state)
  type(state_matrix_t), intent(inout), target :: state
  type(state_iterator_t) :: it
  type(quantum_numbers_t), dimension(:,:), allocatable :: qn
  type(quantum_numbers_t), dimension(:,:), allocatable :: qn_con
  integer, dimension(:), allocatable :: me_index
  integer :: depth, n_me, i, j
  depth = state_matrix_get_depth (state)
  n_me = state_matrix_get_n_matrix_elements (state)
  allocate (qn (depth, n_me))
  allocate (me_index (n_me))
  i = 0
  call state_iterator_init (it, state)
  do while (state_iterator_is_valid (it))
     i = i + 1
     qn(:,i) = state_iterator_get_quantum_numbers (it)
     me_index(i) = state_iterator_get_me_index (it)
     call state_iterator_advance (it)
  end do
  do i = 1, n_me
     call quantum_number_array_make_color_contractions (qn(:,i), qn_con)
     do j = 1, size (qn_con, 2)
        call state_matrix_add_state (state, qn_con(:,j), index = me_index(i))
     end do
  end do
end subroutine state_matrix_add_color_contractions
```

This procedure merges two state matrices of equal depth. For each quantum number (flavor, color, helicity), we take the entry from the first argument where defined, otherwise the second one. (If both are defined, we get an off-diagonal matrix.) The resulting trie combines the information of the input tries in all possible ways. Note that values are ignored, all values in the result are zero.

⟨*State matrices: public*⟩+≡

```
public :: merge_state_matrices
```

⟨*State matrices: procedures*⟩+≡

```
subroutine merge_state_matrices (state1, state2, state3)
  type(state_matrix_t), intent(in), target :: state1, state2
  type(state_matrix_t), intent(out) :: state3
  type(state_iterator_t) :: it1, it2
  type(quantum_numbers_t), dimension(state1%depth) :: qn1, qn2
  if (state1%depth /= state2%depth) then
     call state_matrix_write (state1)
     call state_matrix_write (state2)
     call msg_bug ("State matrices merge impossible: incompatible depths")
  end if
```

```
        call state_matrix_init (state3)
        call state_iterator_init (it1, state1)
        do while (state_iterator_is_valid (it1))
            qn1 = state_iterator_get_quantum_numbers (it1)
            call state_iterator_init (it2, state2)
            do while (state_iterator_is_valid (it2))
                qn2 = state_iterator_get_quantum_numbers (it2)
                call state_matrix_add_state &
                     (state3, qn1 .merge. qn2)
                call state_iterator_advance (it2)
            end do
            call state_iterator_advance (it1)
        end do
        call state_matrix_freeze (state3)
    end subroutine merge_state_matrices
```

Multiply matrix elements from two state matrices. Choose the elements as given by the integer index arrays, multiply them and store the sum of products in the indicated matrix element. The suffixes mean: c=conjugate first factor; f=include weighting factor.

Note that the `dot_product` intrinsic function conjugates its first complex argument. This is intended for the `c` suffix case, but must be reverted for the plain-product case.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_evaluate_product
    public :: state_matrix_evaluate_product_cf
    public :: state_matrix_evaluate_square_c
    public :: state_matrix_evaluate_sum
```

⟨*State matrices: procedures*⟩+≡
```
    pure subroutine state_matrix_evaluate_product &
         (state, i, state1, state2, index1, index2)
      type(state_matrix_t), intent(inout) :: state
      integer, intent(in) :: i
      type(state_matrix_t), intent(in) :: state1, state2
      integer, dimension(:), intent(in) :: index1, index2
      state%me(i) = &
           dot_product (conjg (state1%me(index1)), state2%me(index2))
      state%norm = state1%norm * state2%norm
    end subroutine state_matrix_evaluate_product

    pure subroutine state_matrix_evaluate_product_cf &
         (state, i, state1, state2, index1, index2, factor)
      type(state_matrix_t), intent(inout) :: state
      integer, intent(in) :: i
      type(state_matrix_t), intent(in) :: state1, state2
      integer, dimension(:), intent(in) :: index1, index2
      complex(default), dimension(:), intent(in) :: factor
      state%me(i) = &
           dot_product (state1%me(index1), factor * state2%me(index2))
      state%norm = state1%norm * state2%norm
    end subroutine state_matrix_evaluate_product_cf
```

```
pure subroutine state_matrix_evaluate_square_c (state, i, state1, index1)
  type(state_matrix_t), intent(inout) :: state
  integer, intent(in) :: i
  type(state_matrix_t), intent(in) :: state1
  integer, dimension(:), intent(in) :: index1
  state%me(i) = &
       dot_product (state1%me(index1), state1%me(index1))
  state%norm = abs (state1%norm) ** 2
end subroutine state_matrix_evaluate_square_c

pure subroutine state_matrix_evaluate_sum (state, i, state1, index1)
  type(state_matrix_t), intent(inout) :: state
  integer, intent(in) :: i
  type(state_matrix_t), intent(in) :: state1
  integer, dimension(:), intent(in) :: index1
  state%me(i) = &
       sum (state1%me(index1)) * state1%norm
end subroutine state_matrix_evaluate_sum
```

Outer product (of states and matrix elements):

⟨*State matrices: public*⟩+≡
```
    public :: outer_multiply
```

⟨*State matrices: interfaces*⟩+≡
```
    interface outer_multiply
       module procedure outer_multiply_pair
       module procedure outer_multiply_array
    end interface
```

This procedure constructs the outer product of two state matrices.

⟨*State matrices: procedures*⟩+≡
```
    subroutine outer_multiply_pair (state1, state2, state3)
      type(state_matrix_t), intent(in), target :: state1, state2
      type(state_matrix_t), intent(out) :: state3
      type(state_iterator_t) :: it1, it2
      type(quantum_numbers_t), dimension(state1%depth) :: qn1
      type(quantum_numbers_t), dimension(state2%depth) :: qn2
      type(quantum_numbers_t), dimension(state1%depth+state2%depth) :: qn3
      complex(default) :: val1, val2
      call state_matrix_init (state3, store_values=.true.)
      call state_iterator_init (it1, state1)
      do while (state_iterator_is_valid (it1))
         qn1 = state_iterator_get_quantum_numbers (it1)
         val1 = state_iterator_get_matrix_element (it1)
         call state_iterator_init (it2, state2)
         do while (state_iterator_is_valid (it2))
            qn2 = state_iterator_get_quantum_numbers (it2)
            val2 = state_iterator_get_matrix_element (it2)
            qn3(:state1%depth) = qn1
            qn3(state1%depth+1:) = qn2
            call state_matrix_add_state (state3, qn3, value=val1 * val2)
            call state_iterator_advance (it2)
         end do
```

```
      call state_iterator_advance (it1)
    end do
    call state_matrix_freeze (state3)
  end subroutine outer_multiply_pair
```

This executes the above routine iteratively for an arbitrary number of state matrices.

⟨*State matrices: procedures*⟩+≡

```
    subroutine outer_multiply_array (state_in, state_out)
      type(state_matrix_t), dimension(:), intent(in), target :: state_in
      type(state_matrix_t), intent(out) :: state_out
      type(state_matrix_t), dimension(:), allocatable, target :: state_tmp
      integer :: i, n
      n = size (state_in)
      select case (n)
      case (0)
         call state_matrix_init (state_out)
      case (1)
         state_out = state_in(1)
      case (2)
         call outer_multiply_pair (state_in(1), state_in(2), state_out)
      case default
         allocate (state_tmp (n-2))
         call outer_multiply_pair (state_in(1), state_in(2), state_tmp(1))
         do i = 2, n - 2
            call outer_multiply_pair (state_tmp(i-1), state_in(i+1), state_tmp(i))
         end do
         call outer_multiply_pair (state_tmp(n-2), state_in(n), state_out)
         call state_matrix_final (state_tmp)
      end select
    end subroutine outer_multiply_array
```

7.5.5 Factorization

In physical events, the state matrix is factorized into single-particle state matrices. This is essentially a measurement.

In a simulation, we select one particular branch of the state matrix with a probability that is determined by the matrix elements at the leaves. (This makes sense only if the state matrix represents a squared amplitude.) The selection is based on a (random) value x between 0 and one that is provided as the third argument.

For flavor and color, we select a unique value for each particle. For polarization, we have three options (modes). Option 1 is to drop helicity information altogether and sum over all diagonal helicities. Option 2 is to select a unique diagonal helicity in the same way as flavor and color. Option 3 is, for each particle, to trace over all remaining helicities in order to obtain an array of independent single-particle helicity matrices.

Only branches that match the given quantum-number array qn_in, if present, are considered. For this array, color is ignored.

If the optional correlated_state is provided, it is assigned the correlated density matrix for the selected flavor-color branch, so multi-particle spin corre-

lations remain available even if they are dropped in the single-particle density matrices.

The algorithm is as follows: First, we determine the normalization by summing over all diagonal matrix elements. In a second scan, we select one of the diagonal matrix elements by a cumulative comparison with the normalized random number. In the corresponding quantum number array, we undefine the helicity entries. Then, we scan the third time. For each branch that matches the selected quantum number array (i.e., definite flavor and color, arbitrary helicity), we determine its contribution to any of the single-particle state matrices. The matrix-element value is added if all other quantum numbers are diagonal, while the helicity of the chosen particle may be arbitrary; this helicity determines the branch in the single-particle state.

As a result, flavor and color quantum numbers are selected with the correct probability. Within this subset of states, each single-particle state matrix results from tracing over all other particles. Note that the single-particle state matrices are not normalized.

⟨*State matrices: parameters*⟩≡
```
integer, parameter, public :: FM_IGNORE_HELICITY = 1
integer, parameter, public :: FM_SELECT_HELICITY = 2
integer, parameter, public :: FM_FACTOR_HELICITY = 3
```

⟨*State matrices: public*⟩+≡
```
public :: state_matrix_factorize
```

⟨*State matrices: procedures*⟩+≡
```
subroutine state_matrix_factorize &
    (state, mode, x, single_state, correlated_state, qn_in)
  type(state_matrix_t), intent(in), target :: state
  integer, intent(in) :: mode
  real(default), intent(in) :: x
  type(state_matrix_t), &
      dimension(:), allocatable, intent(out) :: single_state
  type(state_matrix_t), intent(out), optional :: correlated_state
  type(quantum_numbers_t), dimension(:), intent(in), optional :: qn_in
  type(state_iterator_t) :: it
  real(default) :: s, xt
  complex(default) :: value
  integer :: i, depth
  type(quantum_numbers_t), dimension(:), allocatable :: qn, qn1
  type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
  logical, dimension(:), allocatable :: diagonal
  logical, dimension(:,:), allocatable :: mask
  if (x /= 0) then
     xt = x * state_matrix_trace (state, qn_in)
  else
     xt = 0
  end if
  s = 0
  depth = state_matrix_get_depth (state)
  allocate (qn (depth), qn1 (depth), diagonal (depth))
  call state_iterator_init (it, state)
  do while (state_iterator_is_valid (it))
     qn = state_iterator_get_quantum_numbers (it)
```

```
   if (present (qn_in)) then
      if (.not. all (qn .fhmatch. qn_in)) then
         call state_iterator_advance (it); cycle
      end if
   end if
   if (all (quantum_numbers_are_diagonal (qn))) then
      value = state_iterator_get_matrix_element (it)
      if (real (value, default) < 0) then
         print *, value
         call msg_bug ("Event generation: " &
               // "Negative real part of matrix element value")
         value = 0
      end if
      s = s + value
      if (s > xt)  exit
   end if
   call state_iterator_advance (it)
end do
if (.not. state_iterator_is_valid (it)) then
   if (s == 0) then
      call state_matrix_write (state)
      call msg_bug ("Event factorization called for zero matrix element")
   else
      call state_iterator_init (it, state)
   end if
end if
allocate (single_state (depth))
call state_matrix_init (single_state, store_values=.true.)
if (present (correlated_state)) &
     call state_matrix_init (correlated_state, store_values=.true.)
qn = state_iterator_get_quantum_numbers (it)
select case (mode)
case (FM_SELECT_HELICITY)  ! single branch selected; shortcut
   do i = 1, depth
      call state_matrix_add_state (single_state(i), &
            (/qn(i)/), value=value)
   end do
   if (.not. present (correlated_state)) then
      call state_matrix_freeze (single_state)
      return
   end if
end select
allocate (qn_mask (depth))
call quantum_numbers_mask_init (qn_mask, .false., .false., .false., .true.)
call quantum_numbers_undefine (qn, qn_mask)
select case (mode)
case (FM_FACTOR_HELICITY)
   allocate (mask (depth, depth))
   mask = .false.
   forall (i = 1:depth)  mask(i,i) = .true.
end select
call state_iterator_init (it, state)
do while (state_iterator_is_valid (it))
   qn1 = state_iterator_get_quantum_numbers (it)
```

```
          if (all (qn .match. qn1)) then
             diagonal = quantum_numbers_are_diagonal (qn1)
             value = state_iterator_get_matrix_element (it)
             select case (mode)
             case (FM_IGNORE_HELICITY)  ! trace over diagonal states that match qn
                if (all (diagonal)) then
                   do i = 1, depth
                      call state_matrix_add_state (single_state(i), &
                           (/qn(i)/), value=value, sum_values=.true.)
                   end do
                end if
             case (FM_FACTOR_HELICITY)  ! trace over all other particles
                do i = 1, depth
                   if (all (diagonal .or. mask(:,i))) then
                      call state_matrix_add_state (single_state(i), &
                           (/qn1(i)/), value=value, sum_values=.true.)
                   end if
                end do
             end select
             if (present (correlated_state)) &
                  call state_matrix_add_state (correlated_state, qn1, value=value)
          end if
          call state_iterator_advance (it)
       end do
     call state_matrix_freeze (single_state)
     if (present (correlated_state)) &
          call state_matrix_freeze (correlated_state)
   end subroutine state_matrix_factorize
```

7.5.6 Test

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_test
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_test ()
  !    print *, "State matrix test 1"
  !    call state_matrix_test1 ()
  !    print *
  !    print *, "State matrix test 2"
  !    call state_matrix_test2 ()
     print *
     print *, "State matrix test 3"
     call state_matrix_test3 ()
    end subroutine state_matrix_test
```

Create two quantum states of equal depth and merge them.

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_test1 ()
      type(state_matrix_t) :: state1, state2, state3
      type(flavor_t), dimension(3) :: flv
      type(color_t), dimension(3) :: col
```

```
    type(helicity_t), dimension(3) :: hel
    type(quantum_numbers_t), dimension(3) :: qn
    call state_matrix_init (state1)
    call flavor_init (flv, (/ 1, 2, 11 /))
    call helicity_init (hel, (/ 1, 1, 1 /))
    call quantum_numbers_init (qn, flv, hel)
    call state_matrix_add_state (state1, qn)
    call helicity_init (hel, (/ 1, 1, 1 /), (/ -1, 1, -1/))
    call quantum_numbers_init (qn, flv, hel)
    call state_matrix_add_state (state1, qn)
    call state_matrix_freeze (state1)
    call state_matrix_write (state1)
    print *
    call state_matrix_init (state2)
    call color_init (col(1), (/  501 /))
    call color_init (col(2), (/ -501 /))
    call color_init (col(3), (/ 0 /))
    call helicity_init (hel, (/ -1, -1, 0 /))
    call quantum_numbers_init (qn, col, hel)
    call state_matrix_add_state (state2, qn)
    call color_init (col(3), (/ 99 /))
    call helicity_init (hel, (/ -1, -1, 0 /))
    call quantum_numbers_init (qn, col, hel)
    call state_matrix_add_state (state2, qn)
    call state_matrix_freeze (state2)
    call state_matrix_write (state2)
    print *
    call merge_state_matrices (state1, state2, state3)
    call state_matrix_write (state3)
    print *
    call state_matrix_collapse (state3, &
         new_quantum_numbers_mask (.false., .false., &
                                   (/.true.,.false.,.false./)))
    call state_matrix_write (state3)
    call state_matrix_final (state1)
    call state_matrix_final (state2)
    call state_matrix_final (state3)
  end subroutine state_matrix_test1
```

Create a correlated three-particle state matrix and factorize it.

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_matrix_test2
    type(state_matrix_t) :: state
    type(state_matrix_t), dimension(:), allocatable :: single_state
    type(state_matrix_t) :: correlated_state
    complex(default) :: u, val
    complex(default), dimension(-1:1) :: v
    integer :: f, h11, h12, h21, h22, i, mode
    type(flavor_t), dimension(2) :: flv
    type(color_t), dimension(2) :: col
    type(helicity_t), dimension(2) :: hel
    type(quantum_numbers_t), dimension(2) :: qn
    u = 1 / 2._default
    v(-1) = (0.6_default, 0._default)
```

```
    v( 1) = (0._default, 0.8_default)
    call state_matrix_init (state)
    do f = 1, 2
       do h11 = -1, 1, 2
          do h12 = -1, 1, 2
             do h21 = -1, 1, 2
                do h22 = -1, 1, 2
                   call flavor_init (flv, (/f, -f/))
                   call color_init (col(1), (/ 1/))
                   call color_init (col(2), (/-1/))
                   call helicity_init (hel, (/h11,h12/), (/h21, h22/))
                   call quantum_numbers_init (qn, flv, col, hel)
                   val = u * v(h11) * v(h12) * conjg (v(h21) * v(h22))
                   call state_matrix_add_state (state, qn)
                end do
             end do
          end do
       end do
    end do
    call state_matrix_freeze (state)
    call state_matrix_write (state)
    print *, "trace = ", state_matrix_trace (state)
    do mode = 1, 3
       print *
       print *, "Mode = ", mode
       call state_matrix_factorize &
            (state, mode, 0.15_default, single_state, correlated_state)
       do i = 1, size (single_state)
          print *
          call state_matrix_write (single_state(i))
          print *, "trace = ", state_matrix_trace (single_state(i))
       end do
       print *
       call state_matrix_write (correlated_state)
       print *, "trace = ", state_matrix_trace (correlated_state)
       call state_matrix_final (single_state)
       call state_matrix_final (correlated_state)
    end do
    call state_matrix_final (state)
  end subroutine state_matrix_test2
```

Create a colored state matrix and add color contractions.

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_matrix_test3
    type(state_matrix_t) :: state
    type(flavor_t), dimension(4) :: flv
    type(color_t), dimension(4) :: col
    type(quantum_numbers_t), dimension(4) :: qn
    call state_matrix_init (state)
    call flavor_init (flv, &
         (/ 1, -HADRON_REMNANT_TRIPLET, -1, HADRON_REMNANT_TRIPLET /))
    call color_init (col(1), (/17/))
    call color_init (col(2), (/-17/))
    call color_init (col(3), (/-19/))
```

```
      call color_init (col(4), (/19/))
      call quantum_numbers_init (qn, flv=flv, col=col)
      call state_matrix_add_state (state, qn)
      call flavor_init (flv, &
           (/ 1, -HADRON_REMNANT_TRIPLET, 21, HADRON_REMNANT_OCTET /))
      call color_init (col(1), (/17/))
      call color_init (col(2), (/-17/))
      call color_init (col(3), (/3, -5/))
      call color_init (col(4), (/5, -3/))
      call quantum_numbers_init (qn, flv=flv, col=col)
      call state_matrix_add_state (state, qn)
      call state_matrix_freeze (state)
      print *, "State:"
      call state_matrix_write (state)
      call state_matrix_add_color_contractions (state)
      print *, "State with contractions:"
      call state_matrix_write (state)
      call state_matrix_final (state)
    end subroutine state_matrix_test3
```

7.6 Interactions

This module defines the `interaction_t` type. It is an extension of the `state_matrix_t` type.

The state matrix is a representation of a multi-particle density matrix. It implements all possible flavor, color, and quantum-number assignments of the entries in a generic density matrix, and it can hold a complex matrix element for each entry. (Note that this matrix can hold non-diagonal entries in color and helicity space.) The `interaction_t` object associates this with a list of momenta, such that the whole object represents a multi-particle state.

The `interaction_t` holds information about which particles are incoming, virtual (i.e., kept for the records), or outgoing. Each particle can be associated to a source within another interaction. This allows us to automatically fill those interaction momenta which have been computed or defined elsewhere. It also contains internal parent-child relations and flags for (virtual) particles which are to be treated as resonances.

A quantum-number mask array summarizes, for each particle within the interaction, the treatment of flavor, color, or helicity (expose or ignore). A list of locks states which particles are bound to have an identical quantum-number mask. This is useful when the mask is changed at one place.

⟨interactions.f90⟩≡
  ⟨*File header*⟩

```
  module interactions
```

  ⟨*Use kinds*⟩
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use prt_lists
```

```
    use expressions
    use flavors
    use colors
    use helicities
    use quantum_numbers
    use state_matrices
```

⟨*Standard module head*⟩

⟨*Interactions: public*⟩

⟨*Interactions: types*⟩

⟨*Interactions: interfaces*⟩

```
contains
```

⟨*Interactions: procedures*⟩

```
end module interactions
```

7.6.1   External interaction links

Each particle in an interaction can have a link to a corresponding particle in another interaction. This allows to fetch the momenta of incoming or virtual particles from the interaction where they are defined. The link object consists of a pointer to the interaction and an index.

⟨*Interactions: types*⟩≡
```
    type :: external_link_t
      private
      type(interaction_t), pointer :: int => null ()
      integer :: i
    end type external_link_t
```

Set an external link.

⟨*Interactions: procedures*⟩≡
```
    subroutine external_link_set (link, int, i)
      type(external_link_t), intent(out) :: link
      type(interaction_t), target, intent(in) :: int
      integer, intent(in) :: i
      if (i /= 0) then
         link%int => int
         link%i = i
      end if
    end subroutine external_link_set
```

Reassign an external link to a new interaction (which should be an image of the original target).

⟨*Interactions: procedures*⟩+≡
```
    subroutine external_link_reassign (link, int_src, int_target)
      type(external_link_t), intent(inout) :: link
      type(interaction_t), intent(in) :: int_src
```

```
       type(interaction_t), intent(in), target :: int_target
       if (associated (link%int)) then
          if (link%int%tag == int_src%tag)  link%int => int_target
       end if
    end subroutine external_link_reassign
```

Return true if the link is set

⟨*Interactions: procedures*⟩+≡
```
    function external_link_is_set (link) result (flag)
      logical :: flag
      type(external_link_t), intent(in) :: link
      flag = associated (link%int)
    end function external_link_is_set
```

Return the interaction pointer.

⟨*Interactions: procedures*⟩+≡
```
    function external_link_get_ptr (link) result (int)
      type(interaction_t), pointer :: int
      type(external_link_t), intent(in) :: link
      int => link%int
    end function external_link_get_ptr
```

Return the index within that interaction

⟨*Interactions: procedures*⟩+≡
```
    function external_link_get_index (link) result (i)
      integer :: i
      type(external_link_t), intent(in) :: link
      i = link%i
    end function external_link_get_index
```

Return a pointer to the momentum of the corresponding particle. If there is no
association, return a null pointer.

⟨*Interactions: procedures*⟩+≡
```
    function external_link_get_momentum_ptr (link) result (p)
      type(vector4_t), pointer :: p
      type(external_link_t), intent(in) :: link
      if (associated (link%int)) then
         p => link%int%p(link%i)
      else
         p => null ()
      end if
    end function external_link_get_momentum_ptr
```

7.6.2   Internal relations

In addition to the external links, particles within the interaction have parent-
child relations. Here, more than one link is possible, and we set up a list.

⟨*Interactions: types*⟩+≡
```
    type :: internal_link_t
      private
```

```
      integer :: i
      type(internal_link_t), pointer :: next => null ()
   end type internal_link_t

   type :: internal_link_list_t
      private
      integer :: length = 0
      type(internal_link_t), pointer :: first => null ()
      type(internal_link_t), pointer :: last => null ()
   end type internal_link_list_t
```

Add an internal link.

⟨*Interactions: procedures*⟩+≡
```
   subroutine internal_link_list_append (link_list, i)
     type(internal_link_list_t), intent(inout) :: link_list
     integer, intent(in) :: i
     type(internal_link_t), pointer :: current
     allocate (current)
     current%i = i
     if (associated (link_list%first)) then
        link_list%last%next => current
     else
        link_list%first => current
     end if
     link_list%last => current
     link_list%length = link_list%length + 1
   end subroutine internal_link_list_append
```

Finalize the list of internal links.

⟨*Interactions: procedures*⟩+≡
```
   subroutine internal_link_list_final (link_list)
      type(internal_link_list_t), intent(inout) :: link_list
      type(internal_link_t), pointer :: current
      do while (associated (link_list%first))
         current => link_list%first
         link_list%first => current%next
         deallocate (current)
      end do
      link_list%last => null ()
      link_list%length = 0
    end subroutine internal_link_list_final
```

We need a deep copy of this list when we assign interaction objects.

⟨*Interactions: interfaces*⟩≡
```
   interface assignment(=)
      module procedure internal_link_list_assign
   end interface
```

⟨*Interactions: procedures*⟩+≡
```
   subroutine internal_link_list_assign (link_list_out, link_list_in)
      type(internal_link_list_t), intent(in) :: link_list_in
      type(internal_link_list_t), intent(out) :: link_list_out
```

```
        type(internal_link_t), pointer :: current, copy
        current => link_list_in%first
        do while (associated (current))
            allocate (copy)
            copy%i = current%i
            if (associated (link_list_out%first)) then
                link_list_out%last%next => copy
            else
                link_list_out%first => copy
            end if
            link_list_out%last => copy
            current => current%next
        end do
    end subroutine internal_link_list_assign
```

Return true if the link list is nonempty:

⟨*Interactions: procedures*⟩+≡
```
    function internal_link_list_has_entries (link_list) result (flag)
      logical :: flag
      type(internal_link_list_t), intent(in) :: link_list
      flag = associated (link_list%first)
    end function internal_link_list_has_entries
```

Return a pointer to the first entry:

⟨*Interactions: procedures*⟩+≡
```
    function internal_link_list_get_first_ptr (link_list) result (link)
      type(internal_link_list_t), intent(in) :: link_list
      type(internal_link_t), pointer :: link
      link => link_list%first
    end function internal_link_list_get_first_ptr
```

Advance this pointer.

⟨*Interactions: procedures*⟩+≡
```
    subroutine internal_link_advance (link)
      type(internal_link_t), pointer :: link
      link => link%next
    end subroutine internal_link_advance
```

Return the index.

⟨*Interactions: procedures*⟩+≡
```
    function internal_link_get_index (link) result (i)
      integer :: i
      type(internal_link_t), intent(in) :: link
      i = link%i
    end function internal_link_get_index
```

Return the list length

⟨*Interactions: procedures*⟩+≡
```
    function internal_link_list_get_length (link_list) result (length)
      integer :: length
      type(internal_link_list_t), intent(in) :: link_list
```

```
      length = link_list%length
    end function internal_link_list_get_length
```

7.6.3 The interaction type

An interaction is an entangled system of particles. Thus, the interaction object consists of two parts: the particle list, and the quantum state which technically is a trie. The subnode levels beyond the trie root node are in correspondence to the particle list, so both should be traversed in parallel.

The particle list is implemented as an allocatable array of four-momenta. The first `n_in` particles are incoming, `n_vir` particles in-between can be kept for bookkeeping, and the last `n_out` particles are outgoing.

Distinct interactions are linked by their particles: for each particle, we have the possibility of links to corresponding particles in other interactions. Furthermore, for bookkeeping purposes we have a self-link array `relations` where the parent-child relations are kept, and a flag array `resonant` which is set for an intermediate resonance.

Each momentum is associated with masks for flavor, color, and helicity. If a mask entry is set, the associated quantum number is to be ignored for that particle. If any mask has changed, the flag `update` is set.

We can have particle pairs locked together. If this is the case, the corresponding mask entries are bound to be equal. This is useful for particles that go through the interaction.

The interaction tag serves bookkeeping purposes. In particular, it identifies links in printout.

⟨*Interactions: public*⟩≡
```
    public :: interaction_t
```

⟨*Interactions: types*⟩+≡
```
    type :: interaction_t
       private
       integer :: tag = 0
       type(state_matrix_t) :: state_matrix
       integer :: n_in = 0
       integer :: n_vir = 0
       integer :: n_out = 0
       integer :: n_tot = 0
       logical, dimension(:), allocatable :: p_is_known
       type(vector4_t), dimension(:), allocatable :: p
       type(external_link_t), dimension(:), allocatable :: source
       type(internal_link_list_t), dimension(:), allocatable :: parents
       type(internal_link_list_t), dimension(:), allocatable :: children
       logical, dimension(:), allocatable :: resonant
       type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
       integer, dimension(:), allocatable :: lock
       logical :: update_state_matrix = .false.
       logical :: update_values = .false.
    end type interaction_t
```

Initialize the particle array with a fixed size. The first `n_in` particles are incoming, the rest outgoing. Masks are optional. There is also an optional tag. The interaction still needs fixing the values, but that is to be done after all branches have been added.

Interaction tags are assigned consecutively, using a `saved` variable local to this procedure. If desired, we can provide a seed for the interaction tags. Such a seed should be positive. The default seed is one. `tag=0` indicates an empty interaction.

If `set_relations` is set and true, we establish parent-child relations for all incoming and outgoing particles. Virtual particles are skipped; this option is normally used only for interations without virtual particles.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_init
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_init &
         (int, n_in, n_vir, n_out, &
          tag, resonant, mask, lock, set_relations, store_values)
      type(interaction_t), intent(out) :: int
      integer, intent(in) :: n_in, n_vir, n_out
      integer, intent(in), optional :: tag
      logical, dimension(:), intent(in), optional :: resonant
      type(quantum_numbers_mask_t), dimension(:), intent(in), optional :: mask
      integer, dimension(:), intent(in), optional :: lock
      logical, intent(in), optional :: set_relations, store_values
      logical :: set_rel
      integer :: i, j
      set_rel = .false.;  if (present (set_relations))  set_rel = set_relations
      call interaction_set_tag (int, tag)
      call state_matrix_init (int%state_matrix, store_values)
      int%n_in = n_in
      int%n_vir = n_vir
      int%n_out = n_out
      int%n_tot = n_in + n_vir + n_out
      allocate (int%p_is_known (int%n_tot))
      int%p_is_known = .false.
      allocate (int%p (int%n_tot))
      allocate (int%source (int%n_tot))
      allocate (int%parents (int%n_tot))
      allocate (int%children (int%n_tot))
      allocate (int%resonant (int%n_tot))
      if (present (resonant)) then
         int%resonant = resonant
      else
         int%resonant = .false.
      end if
      allocate (int%mask (int%n_tot))
      allocate (int%lock (int%n_tot))
      if (present (mask)) then
         int%mask = mask
      end if
      if (present (lock)) then
         int%lock = lock
      else
```

```
        int%lock = 0
     end if
     int%update_state_matrix = .false.
     int%update_values = .true.
     if (set_rel) then
        do i = 1, n_in
           do j = 1, n_out
              call interaction_relate (int, i, n_in + j)
           end do
        end do
     end if
  end subroutine interaction_init
```

Set or create a unique tag for the interaction.

⟨*Interactions: procedures*⟩+≡
```
  subroutine interaction_set_tag (int, tag)
    type(interaction_t), intent(inout) :: int
    integer, intent(in), optional :: tag
    integer, save :: stored_tag = 1
    if (present (tag)) then
       int%tag = tag
    else
       int%tag = stored_tag
       stored_tag = stored_tag + 1
    end if
  end subroutine interaction_set_tag
```

Finalizer: The state-matrix object contains pointers.

⟨*Interactions: public*⟩+≡
```
  public :: interaction_final
```

⟨*Interactions: procedures*⟩+≡
```
  elemental subroutine interaction_final (int)
    type(interaction_t), intent(inout) :: int
    call state_matrix_final (int%state_matrix)
  end subroutine interaction_final
```

Output. The `verbose` option refers to the state matrix output.

⟨*Interactions: public*⟩+≡
```
  public :: interaction_write
```

⟨*Interactions: procedures*⟩+≡
```
  subroutine interaction_write &
       (int, unit, verbose, show_momentum_sum, show_mass)
    type(interaction_t), intent(in) :: int
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
    integer :: u
    integer :: i, index_link
    type(internal_link_t), pointer :: link
    type(interaction_t), pointer :: int_link
    u = output_unit (unit);  if (u < 0)  return
    if (int%tag /= 0) then
```

```
write (u, *)  "Interaction:", int%tag
do i = 1, int%n_tot
   if (i == 1 .and. int%n_in > 0) then
      write (u, *) "Incoming:"
   else if (i == int%n_in + 1 .and. int%n_vir > 0) then
      write (u, *) "Virtual:"
   else if (i == int%n_in + int%n_vir + 1 .and. int%n_out > 0) then
      write (u, *) "Outgoing:"
   end if
   write (u, "(1x,A,1x,I0)", advance="no") "Particle", i
   if (allocated (int%resonant)) then
      if (int%resonant(i)) then
         write (u, *) "[r]"
      else
         write (u, *)
      end if
   else
      write (u, *)
   end if
   if (allocated (int%p)) then
      if (int%p_is_known(i)) then
         call vector4_write (int%p(i), u, show_mass)
      else
         write (u, *)  "  [momentum undefined]"
      end if
   else
      write (u, *) "  [momentum not allocated]"
   end if
   if (allocated (int%mask)) then
      write (u, "(1x,A)", advance="no")  "mask [fch] = "
      call quantum_numbers_mask_write (int%mask(i), u)
      write (u, *)
   end if
   if (internal_link_list_has_entries (int%parents(i)) &
        .or. internal_link_list_has_entries (int%children(i))) then
      write (u, "(1x,A)", advance="no") "internal links:"
      link => internal_link_list_get_first_ptr (int%parents(i))
      do while (associated (link))
         write (u, "(1x,I0)", advance="no") &
              internal_link_get_index (link)
         call internal_link_advance (link)
      end do
      if (internal_link_list_has_entries (int%parents(i))) &
           write (u, "(1x,A)", advance="no") "=>"
      write (u, "(1x,A)", advance="no") "X"
      if (internal_link_list_has_entries (int%children(i))) &
           write (u, "(1x,A)", advance="no") "=>"
      link => internal_link_list_get_first_ptr (int%children(i))
      do while (associated (link))
         write (u, "(1x,I0)", advance="no") &
              internal_link_get_index (link)
         call internal_link_advance (link)
      end do
      write (u, *)
```

```
            end if
            if (allocated (int%lock)) then
               if (int%lock(i) /= 0) then
                  write (u, "(1x,A,1x,I0)")  "lock:", int%lock(i)
               end if
            end if
            if (external_link_is_set (int%source(i))) then
               write (u, "(1x,A)", advance="no") "source:"
               int_link => external_link_get_ptr (int%source(i))
               index_link = external_link_get_index (int%source(i))
               write (u, "(1x,'(',I0,')',I0)", advance="no") &
                    int_link%tag, index_link
               write (u, *)
            end if
         end do
         if (present (show_momentum_sum)) then
            if (allocated (int%p) .and. show_momentum_sum) then
               write (u, *) "Incoming particles (sum):"
               call vector4_write &
                    (sum (int%p(1:int%n_in)), u, show_mass)
               write (u, *) "Outgoing particles (sum):"
               call vector4_write &
                    (sum (int%p(int%n_in+int%n_vir+1:)), u, show_mass)
               write (u, *)
            end if
         end if
         call state_matrix_write (int%state_matrix, &
              write_value_list=verbose, verbose=verbose, unit=unit)
      else
         write (u, *) "Interaction: [empty]"
      end if
    end subroutine interaction_write
```

Assignment: We implement this as a deep copy. This applies, in particular, to the state-matrix and internal-link components. Furthermore, the new interaction acquires a new tag.

⟨*Interactions: public*⟩+≡
```
    public :: assignment(=)
```

⟨*Interactions: interfaces*⟩+≡
```
    interface assignment(=)
       module procedure interaction_assign
    end interface
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_assign (int_out, int_in)
      type(interaction_t), intent(out) :: int_out
      type(interaction_t), intent(in), target :: int_in
      call interaction_set_tag (int_out)
      int_out%state_matrix = int_in%state_matrix
      int_out%n_in  = int_in%n_in
      int_out%n_out = int_in%n_out
      int_out%n_vir = int_in%n_vir
```

```
         int_out%n_tot = int_in%n_tot
         if (allocated (int_in%p_is_known)) then
            allocate (int_out%p_is_known (size (int_in%p_is_known)))
            int_out%p_is_known = int_in%p_is_known
         end if
         if (allocated (int_in%p)) then
            allocate (int_out%p (size (int_in%p)))
            int_out%p = int_in%p
         end if
         if (allocated (int_in%source)) then
            allocate (int_out%source (size (int_in%source)))
            int_out%source = int_in%source
         end if
         if (allocated (int_in%parents)) then
            allocate (int_out%parents (size (int_in%parents)))
            int_out%parents = int_in%parents
         end if
         if (allocated (int_in%children)) then
            allocate (int_out%children (size (int_in%children)))
            int_out%children = int_in%children
         end if
         if (allocated (int_in%resonant)) then
            allocate (int_out%resonant (size (int_in%resonant)))
            int_out%resonant = int_in%resonant
         end if
         if (allocated (int_in%mask)) then
            allocate (int_out%mask (size (int_in%mask)))
            int_out%mask = int_in%mask
         end if
         if (allocated (int_in%lock)) then
            allocate (int_out%lock (size (int_in%lock)))
            int_out%lock = int_in%lock
         end if
         int_out%update_state_matrix = int_in%update_state_matrix
         int_out%update_values = int_in%update_values
      end subroutine interaction_assign
```

7.6.4   Methods inherited from the state matrix member

Until F2003 is standard, we cannot implement inheritance directly. Therefore, we need wrappers for "inherited" methods.

Make a new branch in the state matrix if it does not yet exist. This is not just a wrapper but it introduces the interaction mask: where a quantum number is masked, it is not transferred but set undefined. After this, the value array has to be updated.

⟨*Interactions: public*⟩+≡
```
      public :: interaction_add_state
```

⟨*Interactions: procedures*⟩+≡
```
      subroutine interaction_add_state &
           (int, qn, index, value, sum_values, counter_index, me_index)
        type(interaction_t), intent(inout) :: int
```

```
    type(quantum_numbers_t), dimension(:), intent(in) :: qn
    integer, intent(in), optional :: index
    complex(default), intent(in), optional :: value
    logical, intent(in), optional :: sum_values
    integer, intent(in), optional :: counter_index
    integer, intent(out), optional :: me_index
    type(quantum_numbers_t), dimension(size(qn)) :: qn_tmp
    qn_tmp = qn
    call quantum_numbers_undefine (qn_tmp, int%mask)
    call state_matrix_add_state &
         (int%state_matrix, qn_tmp, index, value, sum_values, &
         counter_index, me_index)
    int%update_values = .true.
  end subroutine interaction_add_state
```

Freeze the quantum state: First collapse the quantum state, i.e., remove quantum numbers if any mask has changed, then fix the array of value pointers.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_freeze
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_freeze (int)
      type(interaction_t), intent(inout) :: int
      if (int%update_state_matrix) then
        call state_matrix_collapse (int%state_matrix, int%mask)
        int%update_state_matrix = .false.
        int%update_values = .true.
      end if
      if (int%update_values) then
        call state_matrix_freeze (int%state_matrix)
        int%update_values = .false.
      end if
    end subroutine interaction_freeze
```

Return true if the state matrix is empty.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_is_empty
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_is_empty (int) result (flag)
      logical :: flag
      type(interaction_t), intent(in) :: int
      flag = state_matrix_is_empty (int%state_matrix)
    end function interaction_is_empty
```

Get the number of values stored in the state matrix:

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_n_matrix_elements
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_n_matrix_elements (int) result (n)
      integer :: n
      type(interaction_t), intent(in) :: int
```

```
    n = state_matrix_get_n_matrix_elements (int%state_matrix)
  end function interaction_get_n_matrix_elements
```

Get the norm of the state matrix (if the norm has been taken out, otherwise this would be unity).

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_norm
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_norm (int) result (norm)
      real(default) :: norm
      type(interaction_t), intent(in) :: int
      norm = state_matrix_get_norm (int%state_matrix)
    end function interaction_get_norm
```

Get the quantum number array that corresponds to a given index.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_quantum_numbers
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_quantum_numbers (int, i) result (qn)
      type(quantum_numbers_t), dimension(:), allocatable :: qn
      type(interaction_t), intent(in), target :: int
      integer, intent(in) :: i
      allocate (qn (state_matrix_get_depth (int%state_matrix)))
      qn = state_matrix_get_quantum_numbers (int%state_matrix, i)
    end function interaction_get_quantum_numbers
```

Get the matrix element that corresponds to a set of quantum numbers, a given index, or return the whole array.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_matrix_element
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_matrix_element (int, i) result (me)
      complex(default) :: me
      type(interaction_t), intent(in) :: int
      integer, intent(in) :: i
      me = state_matrix_get_matrix_element (int%state_matrix, i)
    end function interaction_get_matrix_element
```

Set the complex value(s) stored in the quantum state.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_set_matrix_element
```

⟨*Interactions: interfaces*⟩+≡
```
    interface interaction_set_matrix_element
      module procedure interaction_set_matrix_element_qn
      module procedure interaction_set_matrix_element_all
      module procedure interaction_set_matrix_element_array
      module procedure interaction_set_matrix_element_single
    end interface
```

Indirect access via the quantum number array:

⟨Interactions: procedures⟩+≡
```
    subroutine interaction_set_matrix_element_qn (int, qn, val)
      type(interaction_t), intent(inout) :: int
      type(quantum_numbers_t), dimension(:), intent(in) :: qn
      complex(default), intent(in) :: val
      call state_matrix_set_matrix_element (int%state_matrix, qn, val)
    end subroutine interaction_set_matrix_element_qn
```

Set all entries of the matrix-element array to a given value.

⟨Interactions: procedures⟩+≡
```
    subroutine interaction_set_matrix_element_all (int, value)
      type(interaction_t), intent(inout) :: int
      complex(default), intent(in) :: value
      call state_matrix_set_matrix_element (int%state_matrix, value)
    end subroutine interaction_set_matrix_element_all
```

Set the matrix-element array directly.

⟨Interactions: procedures⟩+≡
```
    subroutine interaction_set_matrix_element_array (int, value)
      type(interaction_t), intent(inout) :: int
      complex(default), dimension(:), intent(in) :: value
      call state_matrix_set_matrix_element (int%state_matrix, value)
    end subroutine interaction_set_matrix_element_array

    pure subroutine interaction_set_matrix_element_single (int, i, value)
      type(interaction_t), intent(inout) :: int
      integer, intent(in) :: i
      complex(default), intent(in) :: value
      call state_matrix_set_matrix_element (int%state_matrix, i, value)
    end subroutine interaction_set_matrix_element_single
```

Renormalize the state matrix by its trace, if nonzero. The renormalization is reflected in the state-matrix norm.

⟨Interactions: public⟩+≡
```
    public :: interaction_normalize_by_trace
```

⟨Interactions: procedures⟩+≡
```
    subroutine interaction_normalize_by_trace (int)
      type(interaction_t), intent(inout) :: int
      call state_matrix_normalize_by_trace (int%state_matrix)
    end subroutine interaction_normalize_by_trace
```

Analogous, but renormalize by maximal (absolute) value.

⟨Interactions: public⟩+≡
```
    public :: interaction_normalize_by_max
```

⟨Interactions: procedures⟩+≡
```
    subroutine interaction_normalize_by_max (int)
      type(interaction_t), intent(inout) :: int
      call state_matrix_normalize_by_max (int%state_matrix)
    end subroutine interaction_normalize_by_max
```

Return the maximum absolute value of color indices.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_max_color_value
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_max_color_value (int) result (cmax)
      integer :: cmax
      type(interaction_t), intent(in) :: int
      cmax = state_matrix_get_max_color_value (int%state_matrix)
    end function interaction_get_max_color_value
```

Factorize the state matrix into single-particle state matrices, the branch selection depending on a (random) value between 0 and 1; optionally also return a correlated state matrix.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_factorize
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_factorize &
        (int, mode, x, single_state, correlated_state, qn_in)
      type(interaction_t), intent(in), target :: int
      integer, intent(in) :: mode
      real(default), intent(in) :: x
      type(state_matrix_t), &
          dimension(:), allocatable, intent(out) :: single_state
      type(state_matrix_t), intent(out), optional :: correlated_state
      type(quantum_numbers_t), dimension(:), intent(in), optional :: qn_in
      call state_matrix_factorize &
          (int%state_matrix, mode, x, single_state, correlated_state, qn_in)
    end subroutine interaction_factorize
```

Sum all matrix element values

⟨*Interactions: public*⟩+≡
```
    public :: interaction_sum
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_sum (int) result (value)
      complex(default) :: value
      type(interaction_t), intent(in) :: int
      value = state_matrix_sum (int%state_matrix)
    end function interaction_sum
```

Append new states which are color-contracted versions of the existing states. The matrix element index of each color contraction coincides with the index of its origin, so no new matrix elements are generated. After this operation, no **freeze** must be performed anymore.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_add_color_contractions
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_add_color_contractions (int)
      type(interaction_t), intent(inout) :: int
      call state_matrix_add_color_contractions (int%state_matrix)
    end subroutine interaction_add_color_contractions
```

Multiply matrix elements from two interactions. Choose the elements as given by the integer index arrays, multiply them and store the sum of products in the indicated matrix element. The suffixes mean: c=conjugate first factor; f=include weighting factor.

⟨*Interactions: public*⟩+≡
```
public :: interaction_evaluate_product
public :: interaction_evaluate_product_cf
public :: interaction_evaluate_square_c
public :: interaction_evaluate_sum
```

⟨*Interactions: procedures*⟩+≡
```
pure subroutine interaction_evaluate_product &
     (int, i, int1, int2, index1, index2)
  type(interaction_t), intent(inout) :: int
  integer, intent(in) :: i
  type(interaction_t), intent(in) :: int1, int2
  integer, dimension(:), intent(in) :: index1, index2
  call state_matrix_evaluate_product &
       (int%state_matrix, i, int1%state_matrix, int2%state_matrix, &
        index1, index2)
end subroutine interaction_evaluate_product

pure subroutine interaction_evaluate_product_cf &
     (int, i, int1, int2, index1, index2, factor)
  type(interaction_t), intent(inout) :: int
  integer, intent(in) :: i
  type(interaction_t), intent(in) :: int1, int2
  integer, dimension(:), intent(in) :: index1, index2
  complex(default), dimension(:), intent(in) :: factor
  call state_matrix_evaluate_product_cf &
       (int%state_matrix, i, int1%state_matrix, int2%state_matrix, &
        index1, index2, factor)
end subroutine interaction_evaluate_product_cf

pure subroutine interaction_evaluate_square_c (int, i, int1, index1)
  type(interaction_t), intent(inout) :: int
  integer, intent(in) :: i
  type(interaction_t), intent(in) :: int1
  integer, dimension(:), intent(in) :: index1
  call state_matrix_evaluate_square_c &
       (int%state_matrix, i, int1%state_matrix, index1)
end subroutine interaction_evaluate_square_c

pure subroutine interaction_evaluate_sum (int, i, int1, index1)
  type(interaction_t), intent(inout) :: int
  integer, intent(in) :: i
  type(interaction_t), intent(in) :: int1
  integer, dimension(:), intent(in) :: index1
  call state_matrix_evaluate_sum &
       (int%state_matrix, i, int1%state_matrix, index1)
end subroutine interaction_evaluate_sum
```

7.6.5 Accessing contents

Return the integer tag.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_tag
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_tag (int) result (tag)
      integer :: tag
      type(interaction_t), intent(in) :: int
      tag = int%tag
    end function interaction_get_tag
```

Return the number of particles.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_n_tot
    public :: interaction_get_n_in
    public :: interaction_get_n_vir
    public :: interaction_get_n_out
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_n_tot (int) result (n_tot)
      integer :: n_tot
      type(interaction_t), intent(in) :: int
      n_tot = int%n_tot
    end function interaction_get_n_tot

    function interaction_get_n_in (int) result (n_in)
      integer :: n_in
      type(interaction_t), intent(in) :: int
      n_in = int%n_in
    end function interaction_get_n_in

    function interaction_get_n_vir (int) result (n_vir)
      integer :: n_vir
      type(interaction_t), intent(in) :: int
      n_vir = int%n_vir
    end function interaction_get_n_vir

    function interaction_get_n_out (int) result (n_out)
      integer :: n_out
      type(interaction_t), intent(in) :: int
      n_out = int%n_out
    end function interaction_get_n_out
```

Return a momentum index. The flags specify whether to keep/drop incoming, virtual, or outgoing momenta. Check for illegal values.

⟨*Interactions: procedures*⟩+≡
```
    function idx (int, i, outgoing)
      integer :: idx
      type(interaction_t), intent(in) :: int
      integer, intent(in) :: i
      logical, intent(in), optional :: outgoing
      logical :: in, vir, out
```

```
                if (present (outgoing)) then
                   in  = .not. outgoing
                   vir = .false.
                   out = outgoing
                else
                   in = .true.
                   vir = .true.
                   out = .true.
                end if
                idx = 0
                if (in) then
                   if (vir) then
                      if (out) then
                         if (i <= int%n_tot)  idx = i
                      else
                         if (i <= int%n_in + int%n_vir)  idx = i
                      end if
                   else if (out) then
                      if (i <= int%n_in) then
                         idx = i
                      else if (i <= int%n_in + int%n_out) then
                         idx = int%n_vir + i
                      end if
                   else
                      if (i <= int%n_in)  idx = i
                   end if
                else if (vir) then
                   if (out) then
                      if (i <= int%n_vir + int%n_out)  idx = int%n_in + i
                   else
                      if (i <= int%n_vir)  idx = int%n_in + i
                   end if
                else if (out) then
                   if (i <= int%n_out)  idx = int%n_in + int%n_vir + i
                end if
                if (idx == 0) then
                   call interaction_write (int)
                   print *, i, in, vir, out
                   call msg_bug (" Momentum index is out of range for this interaction")
                end if
             end function idx
```

Return all or just a specific four-momentum.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_momenta
    public :: interaction_get_momentum
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_momenta (int, outgoing) result (p)
      type(vector4_t), dimension(:), allocatable :: p
      type(interaction_t), intent(in) :: int
      logical, intent(in), optional :: outgoing
      integer :: i
      if (present (outgoing)) then
```

```
      if (outgoing) then
         allocate (p (int%n_out))
      else
         allocate (p (int%n_in))
      end if
   else
      allocate (p (int%n_tot))
   end if
   do i = 1, size (p)
      p(i) = int%p(idx (int, i, outgoing))
   end do
end function interaction_get_momenta

function interaction_get_momentum (int, i, outgoing) result (p)
  type(vector4_t) :: p
  type(interaction_t), intent(in) :: int
  integer, intent(in) :: i
  logical, intent(in), optional :: outgoing
  p = int%p(idx (int, i, outgoing))
end function interaction_get_momentum
```

Transfer PDG codes, masses (initalization) and momenta to a predefined particle list. We use the flavor assignment of the first branch in the interaction state matrix. Only incoming and outgoing particles are transferred. Switch momentum sign for incoming particles.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_init_prt_list
    public :: interaction_momenta_to_prt_list
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_init_prt_list (int, prt_list)
      type(interaction_t), intent(in), target :: int
      type(prt_list_t), intent(out) :: prt_list
      type(flavor_t), dimension(:), allocatable :: flv
      integer :: i
      allocate (flv (int%n_tot))
      flv = quantum_numbers_get_flavor (interaction_get_quantum_numbers (int, 1))
      call prt_list_init (prt_list, int%n_in + int%n_out)
      do i = 1, int%n_in
         call prt_list_set_incoming (prt_list, i, &
              flavor_get_pdg (flv(i)), &
              vector4_null, &
              flavor_get_mass (flv(i)) ** 2)
      end do
      do i = 1, int%n_out
         call prt_list_set_outgoing (prt_list, int%n_in+i, &
              flavor_get_pdg (flv(int%n_in+int%n_vir+i)), &
              vector4_null, &
              flavor_get_mass (flv(int%n_in+int%n_vir+i)) ** 2)
      end do
    end subroutine interaction_init_prt_list

    subroutine interaction_momenta_to_prt_list (int, prt_list)
      type(interaction_t), intent(in) :: int
```

```
      type(prt_list_t), intent(inout) :: prt_list
      call prt_list_set_p_incoming &
           (prt_list, - interaction_get_momenta (int, outgoing=.false.))
      call prt_list_set_p_outgoing &
           (prt_list, interaction_get_momenta (int, outgoing=.true.))
    end subroutine interaction_momenta_to_prt_list
```

Return a shallow copy of the state matrix:

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_state_matrix
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_state_matrix (int) result (state)
      type(state_matrix_t) :: state
      type(interaction_t), intent(in) :: int
      state = int%state_matrix
    end function interaction_get_state_matrix
```

Return the array of resonance flags

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_resonance_flags
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_resonance_flags (int) result (resonant)
      type(interaction_t), intent(in) :: int
      logical, dimension(size(int%resonant)) :: resonant
      resonant = int%resonant
    end function interaction_get_resonance_flags
```

Return the quantum-numbers mask (or part of it)

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_mask
```

⟨*Interactions: interfaces*⟩+≡
```
    interface interaction_get_mask
       module procedure interaction_get_mask_all
       module procedure interaction_get_mask_slice
    end interface
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_mask_all (int) result (mask)
      type(interaction_t), intent(in) :: int
      type(quantum_numbers_mask_t), dimension(size(int%mask)) :: mask
      mask = int%mask
    end function interaction_get_mask_all

    function interaction_get_mask_slice (int, index) result (mask)
      type(interaction_t), intent(in) :: int
      integer, dimension(:), intent(in) :: index
      type(quantum_numbers_mask_t), dimension(size(index)) :: mask
      mask = int%mask(index)
    end function interaction_get_mask_slice
```

Compute the invariant mass squared of the incoming particles (if any, otherwise outgoing).

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_s
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_s (int) result (s)
      real(default) :: s
      type(interaction_t), intent(in) :: int
      if (int%n_in /= 0) then
         s = sum (int%p(:int%n_in)) ** 2
      else
         s = sum (int%p(int%n_vir+1:)) ** 2
      end if
    end function interaction_get_s
```

Compute the Lorentz transformation that transforms the incoming particles from the center-of-mass frame to the lab frame where they are given. If the c.m. mass squared is negative or zero, return the identity.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_cm_transformation
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_cm_transformation (int) result (lt)
      type(lorentz_transformation_t) :: lt
      type(interaction_t), intent(in) :: int
      type(vector4_t) :: p_cm
      real(default) :: s
      if (int%n_in /= 0) then
         p_cm = sum (int%p(:int%n_in))
      else
         p_cm = sum (int%p(int%n_vir+1:))
      end if
      s = p_cm ** 2
      if (s > 0) then
         lt = boost (p_cm, sqrt (s))
      else
         lt = identity
      end if
    end function interaction_get_cm_transformation
```

Return flavor, momentum, and position of the first outgoing unstable particle present in the interaction. Note that we need not iterate through the state matrix; if there is an unstable particle, it will be present in all state-matrix entries.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_unstable_particle
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_get_unstable_particle (int, flv, p, i)
      type(interaction_t), intent(in), target :: int
      type(flavor_t), intent(out) :: flv
      type(vector4_t), intent(out) :: p
      integer, intent(out) :: i
```

```
      type(state_iterator_t) :: it
      type(flavor_t), dimension(int%n_tot) :: flv_array
      call state_iterator_init (it, int%state_matrix)
      flv_array = state_iterator_get_flavor (it)
      do i = int%n_in + int%n_vir + 1, int%n_tot
         if (.not. flavor_is_stable (flv_array(i))) then
            flv = flv_array(i)
            p = int%p(i)
            return
         end if
      end do
   end subroutine interaction_get_unstable_particle
```

7.6.6 Modifying contents

Set the quantum numbers mask.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_set_mask
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_set_mask (int, mask)
      type(interaction_t), intent(inout) :: int
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
      int%mask = mask
      int%update_state_matrix = .true.
    end subroutine interaction_set_mask
```

Merge a particular mask entry, respecting a possible lock for this entry. We apply an OR relation, which means that quantum numbers are summed over if either of the two masks requires it.

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_merge_mask_entry (int, i, mask)
      type(interaction_t), intent(inout) :: int
      integer, intent(in) :: i
      type(quantum_numbers_mask_t), intent(in) :: mask
      integer :: ii
      ii = idx (int, i)
      if (int%mask(ii) .neqv. mask) then
         int%mask(ii) = int%mask(ii) .or. mask
         if (int%lock(ii) /= 0)  int%mask(int%lock(ii)) = mask
      end if
      int%update_state_matrix = .true.
    end subroutine interaction_merge_mask_entry
```

Fill the momenta array, do not care about the quantum numbers of particles.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_reset_momenta
    public :: interaction_set_momenta
    public :: interaction_set_momentum
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_reset_momenta (int)
      type(interaction_t), intent(inout) :: int
      int%p = vector4_null
      int%p_is_known = .true.
    end subroutine interaction_reset_momenta

    subroutine interaction_set_momenta (int, p, outgoing)
      type(interaction_t), intent(inout) :: int
      type(vector4_t), dimension(:), intent(in) :: p
      logical, intent(in), optional :: outgoing
      integer :: i, index
      do i = 1, size (p)
          index = idx (int, i, outgoing)
          int%p(index) = p(i)
          int%p_is_known(index) = .true.
      end do
    end subroutine interaction_set_momenta

    subroutine interaction_set_momentum (int, p, i, outgoing)
      type(interaction_t), intent(inout) :: int
      type(vector4_t), intent(in) :: p
      integer, intent(in) :: i
      logical, intent(in), optional :: outgoing
      integer :: index
      index = idx (int, i, outgoing)
      int%p(index) = p
      int%p_is_known(index) = .true.
    end subroutine interaction_set_momentum
```

This more sophisticated version of setting values is used for structure functions, in particular if nontrivial flavor, color, and helicity may be present: set values selectively for the given flavors. If there is more than one flavor, scan the interaction and check for a matching flavor at the specified particle location. If it matches, insert the value that corresponds to this flavor.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_set_flavored_values
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_set_flavored_values (int, value, flv_in, pos)
      type(interaction_t), intent(inout) :: int
      complex(default), dimension(:), intent(in) :: value
      type(flavor_t), dimension(:), intent(in) :: flv_in
      integer, intent(in) :: pos
      type(state_iterator_t) :: it
      type(flavor_t) :: flv
      integer :: i
  !    stop "Procedure disabled due to ifort11.0 problem"
      if (size (value) == 1) then
          call interaction_set_matrix_element (int, value(1))
      else
          call state_iterator_init (it, int%state_matrix)
          do while (state_iterator_is_valid (it))
              flv = state_iterator_get_flavor (it, pos)
```

```
                SCAN_FLV: do i = 1, size (value)
                   if (flv == flv_in(i)) then
                       call state_iterator_set_matrix_element (it, value(i))
                       exit SCAN_FLV
                   end if
                end do SCAN_FLV
                call state_iterator_advance (it)
             end do
        end if
    end subroutine interaction_set_flavored_values
```

7.6.7 Handling Linked interactions

Store relations between corresponding particles within one interaction. The first
particle is the parent, the second one the child. Links are established in both
directions.

These relations have no effect on the propagation of momenta etc., they are
rather used for mother-daughter relations in event output.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_relate
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_relate (int, i1, i2)
      type(interaction_t), intent(inout), target :: int
      integer, intent(in) :: i1, i2
      if (i1 /= 0 .and. i2 /= 0) then
          call internal_link_list_append (int%children(i1), i2)
          call internal_link_list_append (int%parents(i2), i1)
      end if
    end subroutine interaction_relate
```

Transfer relations defined within one interaction to a new interaction where the
particle indices are mapped to. Some outgoing particles may have no image.
In that case, a child entry maps to zero, and we skip this relation. Note that
outgoing particles have no children, so parent entries never map to zero.

Also transfer resonance flags.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_transfer_relations
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_transfer_relations (int1, int2, map)
      type(interaction_t), intent(in) :: int1
      type(interaction_t), intent(inout), target :: int2
      integer, dimension(:), intent(in) :: map
      type(internal_link_t), pointer :: link
      integer :: i, k
      do i = 1, size (map)
         link => internal_link_list_get_first_ptr (int1%parents(i))
         do while (associated (link))
            k = internal_link_get_index (link)
            call interaction_relate (int2, map(k), map(i))
            call internal_link_advance (link)
```

```
      end do
      if (map(i) /= 0) then
          int2%resonant(map(i)) = int1%resonant(i)
      end if
   end do
end subroutine interaction_transfer_relations
```

Make up parent-child relations for the particle(s) that are connected to a new interaction.

We scan the connections. For each out/in pair `k1,k2` that we get, we look at the relations of `k1` within `int1`. For each parent `i1` of `k1`, we relate its image to the image of the in-particle `k2` (which is located in `int2`, but this information is not used).

If `resonant` is defined and true, the connections are marked as resonant in the result interaction

⟨*Interactions: public*⟩+≡
```
public :: interaction_relate_connections
```

⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_relate_connections &
    (int, int_in, connection_index, &
     map, map_connections, resonant)
  type(interaction_t), intent(inout), target :: int
  type(interaction_t), intent(in) :: int_in
  integer, dimension(:), intent(in) :: connection_index
  integer, dimension(:), intent(in) :: map, map_connections
  logical, intent(in), optional :: resonant
  logical :: reson
  integer :: i, i2, k2
  type(internal_link_t), pointer :: link
  reson = .false.;  if (present (resonant))  reson = resonant
  do i = 1, size (map_connections)
     k2 = connection_index(i)
     link => internal_link_list_get_first_ptr (int_in%children(k2))
     do while (associated (link))
        i2 = internal_link_get_index (link)
        call interaction_relate (int, map_connections(i), map(i2))
        call internal_link_advance (link)
     end do
     int%resonant(map_connections(i)) = reson
  end do
end subroutine interaction_relate_connections
```

Return the source/target links of the internal connections of particle `i` as an array.

⟨*Interactions: public*⟩+≡
```
public :: interaction_get_children
public :: interaction_get_parents
```

⟨*Interactions: procedures*⟩+≡
```
function interaction_get_children (int, i) result (idx)
  integer, dimension(:), allocatable :: idx
  type(interaction_t), intent(in) :: int
```

```
      integer, intent(in) :: i
      integer :: k
      type(internal_link_t), pointer :: link
      allocate (idx (internal_link_list_get_length (int%children(i))))
      k = 0
      link => internal_link_list_get_first_ptr (int%children(i))
      do while (associated (link))
         k = k + 1
         idx(k) = internal_link_get_index (link)
         call internal_link_advance (link)
      end do
    end function interaction_get_children

    function interaction_get_parents (int, i) result (idx)
      integer, dimension(:), allocatable :: idx
      type(interaction_t), intent(in) :: int
      integer, intent(in) :: i
      integer :: k
      type(internal_link_t), pointer :: link
      allocate (idx (internal_link_list_get_length (int%parents(i))))
      k = 0
      link => internal_link_list_get_first_ptr (int%parents(i))
      do while (associated (link))
         k = k + 1
         idx(k) = internal_link_get_index (link)
         call internal_link_advance (link)
      end do
    end function interaction_get_parents
```

Add a source link from an interaction to a corresponding particle within another interaction. These links affect the propagation of particles: the two linked particles are considered as the same particle, outgoing and incoming.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_set_source_link
```

⟨*Interactions: interfaces*⟩+≡
```
    interface interaction_set_source_link
       module procedure interaction_set_source_link_int
    end interface
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_set_source_link_int (int, i, int1, i1)
      type(interaction_t), intent(inout) :: int
      integer, intent(in) :: i
      type(interaction_t), intent(in), target :: int1
      integer, intent(in) :: i1
      if (i /= 0)  call external_link_set (int%source(i), int1, i1)
    end subroutine interaction_set_source_link_int
```

Reassign links to a new interaction (which is an image of the current interaction).

⟨*Interactions: public*⟩+≡
```
    public :: interaction_reassign_links
```

⟨Interactions: procedures⟩+≡
```
    subroutine interaction_reassign_links (int, int_src, int_target)
      type(interaction_t), intent(inout) :: int
      type(interaction_t), intent(in) :: int_src
      type(interaction_t), intent(in), target :: int_target
      integer :: i
      if (allocated (int%source)) then
         do i = 1, size (int%source)
            call external_link_reassign (int%source(i), int_src, int_target)
         end do
      end if
    end subroutine interaction_reassign_links
```

Since links are one-directional, if we want to follow them backwards we have
to scan all possibilities. This procedure returns the index of the particle within
`int` which points to the particle `i1` within interaction `int1`. If unsuccessful,
return zero.

⟨Interactions: public⟩+≡
```
    public :: interaction_find_link
```

⟨Interactions: procedures⟩+≡
```
    function interaction_find_link (int, int1, i1) result (i)
      integer :: i
      type(interaction_t), intent(in) :: int, int1
      integer, intent(in) :: i1
      type(interaction_t), pointer :: int_tmp
      do i = 1, int%n_tot
         int_tmp => external_link_get_ptr (int%source(i))
         if (int_tmp%tag == int1%tag) then
            if (external_link_get_index (int%source(i)) == i1)  return
         end if
      end do
      i = 0
    end function interaction_find_link
```

Update mask entries by merging them with corresponding masks in interactions
linked to the current one. The mask determines quantum numbers which are
summed over.

Note that both the mask of the current interaction and the mask of the
linked interaction are updated (side effect!). This ensures that both agree for
the linked particle.

⟨Interactions: public⟩+≡
```
    public :: interaction_exchange_mask
```

⟨Interactions: procedures⟩+≡
```
    subroutine interaction_exchange_mask (int)
      type(interaction_t), intent(inout) :: int
      integer :: i, index_link
      type(interaction_t), pointer :: int_link
      do i = 1, int%n_tot
         if (external_link_is_set (int%source(i))) then
            int_link => external_link_get_ptr (int%source(i))
            index_link = external_link_get_index (int%source(i))
```

```
                  call interaction_merge_mask_entry &
                       (int, i, int_link%mask(index_link))
                  call interaction_merge_mask_entry &
                       (int_link, index_link, int%mask(i))
              end if
         end do
         call interaction_freeze (int)
      end subroutine interaction_exchange_mask
```

Copy momenta from interactions linked to the current one.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_receive_momenta
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_receive_momenta (int)
      type(interaction_t), intent(inout) :: int
      integer :: i, index_link
      type(interaction_t), pointer :: int_link
      do i = 1, int%n_tot
         if (external_link_is_set (int%source(i))) then
            int_link => external_link_get_ptr (int%source(i))
            index_link = external_link_get_index (int%source(i))
            call interaction_set_momentum (int, int_link%p(index_link), i)
         end if
      end do
    end subroutine interaction_receive_momenta
```

7.6.8 Recovering connections

When creating an evaluator for two interactions, we have to know by which
particles they are connected. The connection indices can be determined if we
have two linked interactions. We assume that `int1` is the source and `int2` the
target, so the connections of interest are stored within `int2`. A connection is
found if `int2` has a source within `int1`. The result is an array of index pairs.

Connections may also be present indirectly. This is the case if the source of
`int2` coincides with the source of `int1`.

To make things simple, we scan the interaction twice, once for counting hits,
then allocate the array, then scan again and store the connections.

⟨*Interactions: public*⟩+≡
```
    public :: find_connections
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine find_connections (int1, int2, n, connection_index)
      type(interaction_t), intent(in) :: int1, int2
      integer, intent(out) :: n
      integer, dimension(:,:), intent(out), allocatable :: connection_index
      integer :: i, j, k
      type(interaction_t), pointer :: int_link, int_link1
      n = 0
      do i = 1, size (int2%source)
         if (external_link_is_set (int2%source(i))) then
            int_link => external_link_get_ptr (int2%source(i))
```

```
            if (int_link%tag == int1%tag) then
               n = n + 1
            else
               k = external_link_get_index (int2%source(i))
               do j = 1, size (int1%source)
                  if (external_link_is_set (int1%source(j))) then
                     int_link1 => external_link_get_ptr (int1%source(j))
                     if (int_link1%tag == int_link%tag) then
                        if (external_link_get_index (int1%source(j)) == k) then
                           n = n + 1
                        end if
                     end if
                  end if
               end do
            end if
         end if
      end do
      allocate (connection_index (n, 2))
      n = 0
      do i = 1, size (int2%source)
         if (external_link_is_set (int2%source(i))) then
            int_link => external_link_get_ptr (int2%source(i))
            if (int_link%tag == int1%tag) then
               n = n + 1
               connection_index(n,1) = external_link_get_index (int2%source(i))
               connection_index(n,2) = i
            else
               k = external_link_get_index (int2%source(i))
               do j = 1, size (int1%source)
                  if (external_link_is_set (int1%source(j))) then
                     int_link1 => external_link_get_ptr (int1%source(j))
                     if (int_link1%tag == int_link%tag) then
                        if (external_link_get_index (int1%source(j)) == k) then
                           n = n + 1
                           connection_index(n,1) = j
                           connection_index(n,2) = i
                        end if
                     end if
                  end if
               end do
            end if
         end if
      end do
   end subroutine find_connections
```

7.6.9   Test

Generate an interaction of a polarized virtual photon and a colored quark which
may be either up or down. Remove the quark polarization. Generate another
interaction for the quark radiating a photon and link this to the first interation.
The radiation ignores polarization; transfer this information to the first inter-
action to simplify it. Then, transfer the momentum to the radiating quark and

perform a splitting.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_test
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_test ()
      type(interaction_t), target :: int, rad
      type(vector4_t), dimension(3) :: p
      type(quantum_numbers_mask_t), dimension(3) :: mask
      p(2) = vector4_moving (500._default, 500._default, 1)
      p(3) = vector4_moving (500._default,-500._default, 1)
      p(1) = p(2) + p(3)
      call interaction_init (int, 1, 0, 2, set_relations=.true.)
      call int_set (1, -1, 1, 1, (0.3_default, 0.1_default))
      call int_set (1, -1,-1, 1, (0.5_default,-0.7_default))
      call int_set (1, 1, 1, 1, (0.1_default, 0._default))
      call int_set (-1, 1, -1, 2, (0.4_default, -0.1_default))
      call int_set (1, 1, 1, 2, (0.2_default, 0._default))
      call interaction_freeze (int)
      call interaction_set_momenta (int, p)
      mask = new_quantum_numbers_mask (.false.,.false., (/.true.,.true.,.true./))
      call interaction_init (rad, 1, 0, 2, mask=mask, set_relations=.true.)
      call rad_set (1)
      call rad_set (2)
      call interaction_set_source_link (rad, 1, int, 2)
      call interaction_exchange_mask (rad)
      call interaction_receive_momenta (rad)
      p(1) = interaction_get_momentum (rad, 1)
      p(2) = 0.4_default * p(1)
      p(3) = p(1) - p(2)
      call interaction_set_momenta (rad, p(2:3), outgoing=.true.)
      call interaction_freeze (int)
      call interaction_freeze (rad)
      call interaction_set_matrix_element (rad, (0._default, 0._default))
      call interaction_write (int)
      print *
      call interaction_write (rad)
      call interaction_final (int)
      call interaction_final (rad)
    contains
      subroutine int_set (h1, h2, hq, q, val)
        integer, intent(in) :: h1, h2, hq, q
        type(flavor_t), dimension(3) :: flv
        type(color_t), dimension(3) :: col
        type(helicity_t), dimension(3) :: hel
        type(quantum_numbers_t), dimension(3) :: qn
        complex(default), intent(in) :: val
        call flavor_init (flv, (/21, q, -q/))
        call color_init_col_acl (col(2), 5, 0)
        call color_init_col_acl (col(3), 0, 5)
        call helicity_init (hel, (/h1, hq, -hq/), (/h2, hq, -hq/))
        call quantum_numbers_init (qn, flv, col, hel)
        call interaction_add_state (int, qn)
        call interaction_set_matrix_element (int, qn, val)
      end subroutine int_set
```

624

```
    subroutine rad_set (q)
      integer, intent(in) :: q
      type(flavor_t), dimension(3) :: flv
      type(quantum_numbers_t), dimension(3) :: qn
      call flavor_init (flv, (/ q, q, 21 /))
      call quantum_numbers_init (qn, flv)
      call interaction_add_state (rad, qn)
    end subroutine rad_set
  end subroutine interaction_test
```

7.7 Matrix element evaluation

The `evaluator_t` type is an extension of the `interaction_t` type. It represents either a density matrix as the square of a transition matrix element, or the product of two density matrices. Usually, some quantum numbers are summed over in the result.

The `interaction_t` subobject represents a multi-particle interaction with incoming, virtual, and outgoing particles and the associated (not necessarily diagonal) density matrix of quantum state. When the evaluator is initialized, this interaction is constructed from the input interaction(s).

In addition, the initialization process sets up a multiplication table. For each matrix element of the result, it states which matrix elements are to be taken from the input interaction(s), multiplied (optionally, with an additional weight factor) and summed over.

Eventually, to a processes we associate a chain of evaluators which are to be evaluated sequentially. The physical event and its matrix element value(s) can be extracted from the last evaluator in such a chain.

Evaluators are constructed only once (as long as this is possible) during an initialization step. Then, for each event, momenta are computed and transferred among evaluators using the links within the interaction subobject. The multiplication tables enable fast evaluation of the result without looking at quantum numbers anymore.

⟨`evaluators.f90`⟩≡
  ⟨*File header*⟩

```
  module evaluators
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use models
    use flavors
    use colors
    use helicities
    use quantum_numbers
    use state_matrices
    use interactions
```

⟨*Standard module head*⟩

⟨*Evaluators: public*⟩

⟨*Evaluators: parameters*⟩

⟨*Evaluators: types*⟩

⟨*Evaluators: interfaces*⟩

```
contains
```

⟨*Evaluators: procedures*⟩

```
end module evaluators
```

7.7.1 Array of pairings

The evaluator contains an array of `pairing_array` objects. This makes up the multiplication table.

Each pairing array contains two list of matrix element indices and a list of numerical factors. The matrix element indices correspond to the input interactions. The corresponding matrix elements are to be multiplied and optionally multiplied by a factor. The results are summed over to yield one specific matrix element of the result evaluator.

⟨*Evaluators: types*⟩≡

```
   type :: pairing_array_t
      integer, dimension(:), allocatable :: i1, i2
      complex(default), dimension(:), allocatable :: factor
   end type pairing_array_t
```

⟨*Evaluators: procedures*⟩≡

```
   elemental subroutine pairing_array_init (pa, n, has_i2, has_factor)
     type(pairing_array_t), intent(out) :: pa
     integer, intent(in) :: n
     logical, intent(in) :: has_i2, has_factor
     allocate (pa%i1 (n))
     if (has_i2)  allocate (pa%i2 (n))
     if (has_factor)  allocate (pa%factor (n))
   end subroutine pairing_array_init
```

7.7.2 The evaluator type

Possible variants of evaluators:

⟨*Evaluators: parameters*⟩≡

```
   integer, parameter :: &
        EVAL_UNDEFINED = 0, &
        EVAL_PRODUCT = 1, &
        EVAL_SQUARED_FLOWS = 2, &
        EVAL_SQUARE_WITH_COLOR_FACTORS = 3, &
        EVAL_COLOR_CONTRACTION = 4
```

626

The evaluator type contains the result interaction and an array of pairing lists, one for each matrix element in the result interaction.

⟨*Evaluators: public*⟩≡
```
    public :: evaluator_t
```

⟨*Evaluators: types*⟩+≡
```
    type :: evaluator_t
       private
       integer :: type = EVAL_UNDEFINED
       type(interaction_t), pointer :: int_in1 => null ()
       type(interaction_t), pointer :: int_in2 => null ()
       type(interaction_t) :: int
       type(pairing_array_t), dimension(:), allocatable :: pairing_array
    end type evaluator_t
```

Output.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_write
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_write (eval, unit, verbose, show_momentum_sum, show_mass)
      type(evaluator_t), intent(in) :: eval
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
      logical :: conjugate, square
      integer :: u, i, j
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)")  "Evaluator:"
      call interaction_write &
           (eval%int, unit, verbose, show_momentum_sum, show_mass)
      write (u, "(1x,A)")  "Matrix-element multiplication"
      write (u, "(2x,A)", advance="no")  "Input interaction 1:"
      if (associated (eval%int_in1)) then
         write (u, "(1x,I0)")  interaction_get_tag (eval%int_in1)
      else
         write (u, *)  " [undefined]"
      end if
      write (u, "(2x,A)", advance="no")  "Input interaction 2:"
      if (associated (eval%int_in2)) then
         write (u, *)  interaction_get_tag (eval%int_in2)
      else
         write (u, *)  " [undefined]"
      end if
      select case (eval%type)
      case (EVAL_SQUARED_FLOWS, EVAL_SQUARE_WITH_COLOR_FACTORS)
         conjugate = .true.
         square = .true.
      case default
         conjugate = .false.
         square = .false.
      end select
      if (allocated (eval%pairing_array)) then
         do i = 1, size (eval%pairing_array)
            write (u, "(2x,A,I0,A)")  "ME(", i, ") = "
```

```
                  do j = 1, size (eval%pairing_array(i)%i1)
                     write (u, "(4x,A)", advance="no")  "+"
                     if (allocated (eval%pairing_array(i)%i2)) then
                        write (u, "(1x,A,I0,A)", advance="no")  &
                          "ME1(", eval%pairing_array(i)%i1(j), ")"
                        if (conjugate) then
                           write (u, "(A)", advance="no")  "* x"
                        else
                           write (u, "(A)", advance="no")  " x"
                        end if
                        write (u, "(1x,A,I0,A)", advance="no")  &
                          "ME2(", eval%pairing_array(i)%i2(j), ")"
                     else if (square) then
                        write (u, "(1x,A)", advance="no")  "|"
                        write (u, "(A,I0,A)", advance="no")  &
                          "ME1(", eval%pairing_array(i)%i1(j), ")"
                        write (u, "(A)", advance="no")  "|^2"
                     else
                        write (u, "(1x,A,I0,A)", advance="no")  &
                          "ME1(", eval%pairing_array(i)%i1(j), ")"
                     end if
                     if (allocated (eval%pairing_array(i)%factor)) then
                        write (u, "(1x,A)", advance="no")  "x"
                        write (u, *)  eval%pairing_array(i)%factor(j)
                     else
                        write (u, *)
                     end if
                  end do
               end do
         end if
      end subroutine evaluator_write
```

Assignment: Deep copy of the interaction component.

⟨*Evaluators: public*⟩+≡
```
      public :: assignment(=)
```

⟨*Evaluators: interfaces*⟩≡
```
      interface assignment(=)
         module procedure evaluator_assign
      end interface
```

⟨*Evaluators: procedures*⟩+≡
```
      subroutine evaluator_assign (eval_out, eval_in)
        type(evaluator_t), intent(out) :: eval_out
        type(evaluator_t), intent(in) :: eval_in
        eval_out%type = eval_in%type
        eval_out%int_in1 => eval_in%int_in1
        eval_out%int_in2 => eval_in%int_in2
        eval_out%int = eval_in%int
        if (allocated (eval_in%pairing_array)) then
           allocate (eval_out%pairing_array (size (eval_in%pairing_array)))
           eval_out%pairing_array = eval_in%pairing_array
        end if
      end subroutine evaluator_assign
```

7.7.3 Auxiliary structures for evaluator creation

Creating an evaluator that properly handles all quantum numbers requires some bookkeeping. In this section, we define several auxiliary types and methods that organize and simplify this task. More structures are defined within the specific initializers (as local types and internal subroutines).

These types are currently implemented in a partial object-oriented way: We define some basic methods for initialization etc. here, but the evaluator routines below do access their internals as well. This simplifies some things such as index addressing using array slices, at the expense of losing some clarity.

Index mapping

Index mapping are abundant when constructing an evaluator. To have arrays of index mappings, we define this:

⟨*Evaluators: types*⟩+≡
```
type :: index_map_t
   integer, dimension(:), allocatable :: entry
end type index_map_t
```

⟨*Evaluators: procedures*⟩+≡
```
elemental subroutine index_map_init (map, n)
  type(index_map_t), intent(out) :: map
  integer, intent(in) :: n
  allocate (map%entry (n))
end subroutine index_map_init
```

⟨*Evaluators: procedures*⟩+≡
```
function index_map_exists (map) result (flag)
  logical :: flag
  type(index_map_t), intent(in) :: map
  flag = allocated (map%entry)
end function index_map_exists
```

⟨*Evaluators: interfaces*⟩+≡
```
interface size
   module procedure index_map_size
end interface
```

⟨*Evaluators: procedures*⟩+≡
```
function index_map_size (map) result (s)
  integer :: s
  type(index_map_t), intent(in) :: map
  if (allocated (map%entry)) then
     s = size (map%entry)
  else
     s = 0
  end if
end function index_map_size
```

⟨*Evaluators: interfaces*⟩+≡
```
    interface assignment(=)
       module procedure index_map_assign_int
       module procedure index_map_assign_array
    end interface
```

⟨*Evaluators: procedures*⟩+≡
```
    elemental subroutine index_map_assign_int (map, ival)
      type(index_map_t), intent(inout) :: map
      integer, intent(in) :: ival
      map%entry = ival
    end subroutine index_map_assign_int

    subroutine index_map_assign_array (map, array)
      type(index_map_t), intent(inout) :: map
      integer, dimension(:), intent(in) :: array
      map%entry = array
    end subroutine index_map_assign_array
```

⟨*Evaluators: procedures*⟩+≡
```
    elemental subroutine index_map_set_entry (map, i, ival)
      type(index_map_t), intent(inout) :: map
      integer, intent(in) :: i
      integer, intent(in) :: ival
      map%entry(i) = ival
    end subroutine index_map_set_entry
```

⟨*Evaluators: procedures*⟩+≡
```
    elemental function index_map_get_entry (map, i) result (ival)
      integer :: ival
      type(index_map_t), intent(in) :: map
      integer, intent(in) :: i
      ival = map%entry(i)
    end function index_map_get_entry
```

Index mapping (two-dimensional)

This is a variant with a square matrix instead of an array.

⟨*Evaluators: types*⟩+≡
```
    type :: index_map2_t
       integer :: s = 0
       integer, dimension(:,:), allocatable :: entry
    end type index_map2_t
```

⟨*Evaluators: procedures*⟩+≡
```
    elemental subroutine index_map2_init (map, n)
      type(index_map2_t), intent(out) :: map
      integer, intent(in) :: n
      map%s = n
      allocate (map%entry (n, n))
    end subroutine index_map2_init
```

630

⟨*Evaluators: procedures*⟩+≡
```
function index_map2_exists (map) result (flag)
  logical :: flag
  type(index_map2_t), intent(in) :: map
  flag = allocated (map%entry)
end function index_map2_exists
```

⟨*Evaluators: interfaces*⟩+≡
```
interface size
   module procedure index_map2_size
end interface
```

⟨*Evaluators: procedures*⟩+≡
```
function index_map2_size (map) result (s)
  integer :: s
  type(index_map2_t), intent(in) :: map
  s = map%s
end function index_map2_size
```

⟨*Evaluators: interfaces*⟩+≡
```
interface assignment(=)
   module procedure index_map2_assign_int
end interface
```

⟨*Evaluators: procedures*⟩+≡
```
elemental subroutine index_map2_assign_int (map, ival)
  type(index_map2_t), intent(inout) :: map
  integer, intent(in) :: ival
  map%entry = ival
end subroutine index_map2_assign_int
```

⟨*Evaluators: procedures*⟩+≡
```
elemental subroutine index_map2_set_entry (map, i, j, ival)
  type(index_map2_t), intent(inout) :: map
  integer, intent(in) :: i, j
  integer, intent(in) :: ival
  map%entry(i,j) = ival
end subroutine index_map2_set_entry
```

⟨*Evaluators: procedures*⟩+≡
```
elemental function index_map2_get_entry (map, i, j) result (ival)
  integer :: ival
  type(index_map2_t), intent(in) :: map
  integer, intent(in) :: i, j
  ival = map%entry(i,j)
end function index_map2_get_entry
```

631

Auxiliary structures: particle mask

This is a simple container of a logical array.

⟨*Evaluators: types*⟩+≡
```
type :: prt_mask_t
   logical, dimension(:), allocatable :: entry
end type prt_mask_t
```

⟨*Evaluators: procedures*⟩+≡
```
subroutine prt_mask_init (mask, n)
  type(prt_mask_t), intent(out) :: mask
  integer, intent(in) :: n
  allocate (mask%entry (n))
end subroutine prt_mask_init
```

⟨*Evaluators: interfaces*⟩+≡
```
interface size
   module procedure prt_mask_size
end interface
```

⟨*Evaluators: procedures*⟩+≡
```
function prt_mask_size (mask) result (s)
  integer :: s
  type(prt_mask_t), intent(in) :: mask
  s = size (mask%entry)
end function prt_mask_size
```

Quantum number containers

Trivial transparent containers:

⟨*Evaluators: types*⟩+≡
```
type :: qn_list_t
   type(quantum_numbers_t), dimension(:,:), allocatable :: qn
end type qn_list_t

type :: qn_mask_array_t
   type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
end type qn_mask_array_t
```

Auxiliary structures: connection entries

This type is used as intermediate storage when computing the product of two evaluators or the square of an evaluator. The quantum-number array `qn` corresponds to the particles common to both interactions, but irrelevant quantum numbers (color) masked out. The index arrays `index_in` determine the entries in the input interactions that contribute to this connection. `n_index` is the size of these arrays, and `count` is used while filling the entries. Finally, the quantum-number arrays `qn_in_list` are the actual entries in the input interaction that contribute. In the product case, they exclude the connected quantum numbers.

Each evaluator has its own `connection_table` which contains an array of `connection_entry` objects, but also has stuff that specifically applies to the evaluator type. Hence, we do not generalize the `connection_table_t` type.

The filling procedure `connection_entry_add_state` is specific to the various evaluator types.

⟨*Evaluators: types*⟩+≡
```
type :: connection_entry_t
   type(quantum_numbers_t), dimension(:), allocatable :: qn_conn
   integer, dimension(:), allocatable :: n_index
   integer, dimension(:), allocatable :: count
   type(index_map_t), dimension(:), allocatable :: index_in
   type(qn_list_t), dimension(:), allocatable :: qn_in_list
end type connection_entry_t
```

⟨*Evaluators: procedures*⟩+≡
```
subroutine connection_entry_init &
    (entry, n_count, n_map, qn_conn, count, n_rest)
  type(connection_entry_t), intent(out) :: entry
  integer, intent(in) :: n_count, n_map
  type(quantum_numbers_t), dimension(:), intent(in) :: qn_conn
  integer, dimension(n_count), intent(in) :: count
  integer, dimension(n_count), intent(in) :: n_rest
  integer :: i
  allocate (entry%qn_conn (size (qn_conn)))
  allocate (entry%n_index (n_count))
  allocate (entry%count (n_count))
  allocate (entry%index_in (n_map))
  allocate (entry%qn_in_list (n_count))
  entry%qn_conn = qn_conn
  entry%n_index = count
  entry%count = 0
  if (size (entry%index_in) == size (count)) then
     call index_map_init (entry%index_in, count)
  else
     call index_map_init (entry%index_in, count(1))
  end if
  do i = 1, n_count
    allocate (entry%qn_in_list(i)%qn (n_rest(i), count(i)))
  end do
end subroutine connection_entry_init
```

⟨*Evaluators: procedures*⟩+≡
```
subroutine connection_entry_write (entry, unit)
  type(connection_entry_t), intent(in) :: entry
  integer, intent(in), optional :: unit
  integer :: i, j
  integer :: u
  u = output_unit (unit)
  call quantum_numbers_write (entry%qn_conn, unit)
  write (u, *)
  do i = 1, size (entry%n_index)
     write (u, *)  "Input interaction", i
     do j = 1, entry%n_index(i)
```

```
            if (size (entry%n_index) == size (entry%index_in)) then
               write (u, "(2x,I0,4x,I0,2x)", advance = "no") &
                      j, index_map_get_entry (entry%index_in(i), j)
            else
               write (u, "(2x,I0,4x,I0,2x,I0,2x)", advance = "no") &
                      j, index_map_get_entry (entry%index_in(1), j), &
                         index_map_get_entry (entry%index_in(2), j)
            end if
            call quantum_numbers_write (entry%qn_in_list(i)%qn(:,j), unit)
            write (u, *)
         end do
      end do
   end subroutine connection_entry_write
```

Color handling

For managing color-factor computation, we introduce this local type. The `index`
is the index in the color table that corresponds to a given matrix element index
in the input interaction. The `col` array stores the color assignments in rows.
The `factor` array associates a complex number with each pair of arrays in the
color table. The `factor_is_known` array reveals whether a given factor is known
already or still has to be computed.

⟨*Evaluators: types*⟩+≡

```
   type :: color_table_t
      integer, dimension(:), allocatable :: index
      type(color_t), dimension(:,:), allocatable :: col
      logical, dimension(:,:), allocatable :: factor_is_known
      complex(default), dimension(:,:), allocatable :: factor
   end type color_table_t
```

This is the initializer. We extract the color states from the given state matrices,
establish index mappings between the two states (implemented by the array
`me_index`), make an array of color states, and initialize the color-factor table.
The latter is two-dimensional (includes interference) and not yet filled.

⟨*Evaluators: procedures*⟩+≡

```
   subroutine color_table_init (color_table, state, n_tot)
     type(color_table_t), intent(out) :: color_table
     type(state_matrix_t), intent(in) :: state
     integer, intent(in) :: n_tot
     type(state_iterator_t) :: it
     type(quantum_numbers_t), dimension(:), allocatable :: qn
     type(state_matrix_t) :: state_col
     integer :: index, n_col_state
     allocate (color_table%index &
          (state_matrix_get_n_matrix_elements (state)))
     color_table%index = 0
     allocate (qn (n_tot))
     call state_matrix_init (state_col)
     call state_iterator_init (it, state)
     do while (state_iterator_is_valid (it))
        index = state_iterator_get_me_index (it)
```

```
          call quantum_numbers_init (qn, col = state_iterator_get_color (it))
          call state_matrix_add_state (state_col, qn, &
               me_index = color_table%index(index))
          call state_iterator_advance (it)
       end do
       n_col_state = state_matrix_get_n_matrix_elements (state_col)
       allocate (color_table%col (n_tot, n_col_state))
       call state_iterator_init (it, state_col)
       do while (state_iterator_is_valid (it))
          index = state_iterator_get_me_index (it)
          color_table%col(:,index) = state_iterator_get_color (it)
          call state_iterator_advance (it)
       end do
       call state_matrix_final (state_col)
       allocate (color_table%factor_is_known (n_col_state, n_col_state))
       allocate (color_table%factor (n_col_state, n_col_state))
       color_table%factor_is_known = .false.
    end subroutine color_table_init
```

Output (debugging use):

⟨*Evaluators: procedures*⟩+≡

```
    subroutine color_table_write (color_table, unit)
      type(color_table_t), intent(in) :: color_table
      integer, intent(in), optional :: unit
      integer :: i, j
      integer :: u
      u = output_unit (unit)
      write (u, *) "Color table:"
      if (allocated (color_table%index)) then
         write (u, *) "  Index mapping state => color table:"
         do i = 1, size (color_table%index)
            write (u, "(3x,I0,2x,I0,2x)")  i, color_table%index(i)
         end do
         write (u, *) "  Color table:"
         do i = 1, size (color_table%col, 2)
            write (u, "(3x,I0,2x)", advance = "no")  i
            call color_write (color_table%col(:,i), unit)
            write (u, *)
         end do
         write (u, *) "  Defined color factors:"
         do i = 1, size (color_table%factor, 1)
            do j = 1, size (color_table%factor, 2)
               if (color_table%factor_is_known(i,j)) then
                  write (u, *)  i, j, color_table%factor(i,j)
               end if
            end do
         end do
      end if
    end subroutine color_table_write
```

This subroutine sets color factors, based on information from the hard matrix
element: the list of pairs of color-flow indices (in the basis of the matrix element
code), the list of corresponding factors, and the list of mappings from the matrix

635

element index in the input interaction to the color-flow index in the hard matrix element object.

We first determine the mapping of color-flow indices from the hard matrix element code to the current color table. The mapping could be nontrivial because the latter is derived from iterating over a state matrix, which may return states in non-canonical order. The translation table can be determined because we have, for the complete state matrix, both the mapping to the hard interaction (the input `col_index_hi`) and the mapping to the current color table (the component `color_table%index`).

Once this mapping is known, we scan the list of index pairs `color_flow_index` and translate them to valid color-table index pairs. For this pair, the color factor is set using the corresponding entry in the list `col_factor`.

⟨*Evaluators: procedures*⟩+≡

```fortran
    subroutine color_table_set_color_factors (color_table, &
        col_flow_index, col_factor, col_index_hi)
      type(color_table_t), intent(inout) :: color_table
      integer, dimension(:,:), intent(in) :: col_flow_index
      complex(default), dimension(:), intent(in) :: col_factor
      integer, dimension(:), intent(in) :: col_index_hi
      integer, dimension(:), allocatable :: hi_to_ct
      integer :: n_cflow
      integer :: hi_index, me_index, ct_index, cf_index
      integer, dimension(2) :: hi_index_pair, ct_index_pair
      n_cflow = size (col_index_hi)
      if (size (color_table%index) /= n_cflow) &
          call msg_bug ("Mismatch between hard matrix element and color table")
      allocate (hi_to_ct (n_cflow))
      do me_index = 1, size (color_table%index)
         ct_index = color_table%index(me_index)
         hi_index = col_index_hi(me_index)
         hi_to_ct(hi_index) = ct_index
      end do
      do cf_index = 1, size (col_flow_index, 2)
         hi_index_pair = col_flow_index(:,cf_index)
         ct_index_pair = hi_to_ct(hi_index_pair)
         color_table%factor(ct_index_pair(1), ct_index_pair(2)) = &
             col_factor(cf_index)
         color_table%factor_is_known(ct_index_pair(1), ct_index_pair(2)) = .true.
      end do
    end subroutine color_table_set_color_factors
```

This function returns a color factor, given two indices which point to the matrix elements of the initial state matrix. Internally, we can map them to the corresponding indices in the color table. As a side effect, we store the color factor in the color table for later lookup. (I.e., this function is impure.)

⟨*Evaluators: procedures*⟩+≡

```fortran
    function color_table_get_color_factor (color_table, index1, index2, nc) &
        result (factor)
      real(default) :: factor
      type(color_table_t), intent(inout) :: color_table
      integer, intent(in) :: index1, index2
      integer, intent(in), optional :: nc
```

```
      integer :: i1, i2
!     print *, "compute color factor ", index1, index2
      i1 = color_table%index(index1)
      i2 = color_table%index(index2)
!     print *, "  indices = ", i1, i2
      if (color_table%factor_is_known(i1,i2)) then
         factor = color_table%factor(i1,i2)
!        print *, "  is known : ", factor
      else
         factor = compute_color_factor &
              (color_table%col(:,i1), color_table%col(:,i2), nc)
         color_table%factor(i1,i2) = factor
         color_table%factor_is_known(i1,i2) = .true.
!        print *, "  computed : ", factor
      end if
    end function color_table_get_color_factor
```

7.7.4 Creating an evaluator: Matrix multiplication

The evaluator for matrix multiplication is the most complicated variant.

The initializer takes two input interactions and constructs the result evaluator, which consists of the interaction and the multiplication table for the product (or convolution) of the two. Normally, the input interactions are connected by one or more common particles (e.g., decay, structure function convolution).

In the result interaction, quantum numbers of the connections can be summed over. This is determined by the `qn_mask_conn` argument. The `qn_mask_rest` argument is its analog for the other particles within the result interaction. (E.g., for the trace of the state matrix, all quantum numbers are summed over.) Finally, the `connections_are_resonant` argument tells whether the connecting particles should be marked as resonant in the final event record. This is useful for decays.

The algorithm consists of the following steps:

1. `find_connections`: Find the particles which are connected, i.e., common to both input interactions. Either they are directly linked, or both are linked to a common source.

2. `compute_index_bounds_and_mappings`: Compute the mappings of particle indices from the input interactions to the result interaction. There is a separate mapping for the connected particles.

3. `accumulate_connected_states`: Create an auxiliary state matrix which lists the possible quantum numbers for the connected particles. When building this matrix, count the number of times each assignment is contained in any of the input states and, for each of the input states, record the index of the matrix element within the new state matrix. For the connected particles, reassign color indices such that no color state is present twice in different color-index assignment. Note that helicity assignments of the connected state can be (and will be) off-diagonal, so no spin correlations are lost in decays.

   Do this for both input interactions.

4. `allocate_connection_entries`: Allocate a table of connections. Each table row corresponds to one state in the auxiliary matrix, and to multiple states of the input interactions. It collects all states of the unconnected particles in the two input interactions that are associated with the particular state (quantum-number assignment) of the connected particles.

5. `fill_connection_table`: Fill the table of connections by scanning both input interactions. When copying states, reassign color indices for the unconnected particles such that they match between all involved particle sets (interaction 1, interaction 2, and connected particles).

6. `make_product_interaction`: Scan the table of connections we have just built. For each entry, construct all possible pairs of states of the unconnected particles and combine them with the specific connected-particle state. This is a possible quantum-number assignment of the result interaction. Now mask all quantum numbers that should be summed over, and append this to the result state matrix. Record the matrix element index of the result. We now have the result interaction.

7. `make_pairing_array`: First allocate the pairing array with the number of entries of the result interaction. Then scan the table of connections again. For each entry, record the indices of the matrix elements which have to be multiplied and summed over in order to compute this particular matrix element. This makes up the multiplication table.

8. `record_links`: Transfer all source pointers from the input interactions to the result interaction. Do the same for the internal parent-child relations and resonance assignments. For the connected particles, make up appropriate additional parent-child relations. This allows for fetching momenta from other interactions when a new event is filled, and to reconstruct the event history when the event is analyzed.

After all this is done, for each event, we just have to evaluate the pairing arrays (multiplication tables) in order to compute the result matrix elements in their proper positions. The quantum-number assignments remain fixed from now on.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_init_product
```

⟨*Evaluators: interfaces*⟩+≡
```
    interface evaluator_init_product
       module procedure evaluator_init_product_ii
       module procedure evaluator_init_product_ie
       module procedure evaluator_init_product_ei
       module procedure evaluator_init_product_ee
    end interface
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_init_product_ii &
         (eval, int_in1, int_in2, qn_mask_conn, qn_mask_rest, &
          connections_are_resonant)

      type(evaluator_t), intent(out), target :: eval
```

```fortran
    type(interaction_t), intent(in), target :: int_in1, int_in2
    type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
    type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
    logical, intent(in), optional :: connections_are_resonant

    type(qn_mask_array_t), dimension(2) :: qn_mask_in

    type :: connection_table_t
       integer :: n_conn = 0
       integer, dimension(2) :: n_rest = 0
       integer :: n_tot = 0
       integer :: n_me_conn = 0
       type(state_matrix_t) :: state
       type(index_map_t), dimension(:), allocatable :: index_conn
       type(connection_entry_t), dimension(:), allocatable :: entry
       type(index_map_t) :: index_result
    end type connection_table_t
    type(connection_table_t) :: connection_table

    integer :: n_in, n_vir, n_out, n_tot
    integer, dimension(2) :: n_rest
    integer :: n_conn, n_me_conn

    integer, dimension(:,:), allocatable :: connection_index
    type(index_map_t), dimension(2) :: prt_map_in
    type(index_map_t) :: prt_map_conn
    type(prt_mask_t), dimension(2) :: prt_is_connected
    type(quantum_numbers_mask_t), dimension(:), allocatable :: &
         qn_mask_conn_initial

    integer :: i

    eval%type = EVAL_PRODUCT
    eval%int_in1 => int_in1
    eval%int_in2 => int_in2
!    print *, "Evaluator product"
!    print *, "First interaction"
!    call interaction_write (int_in1)
!    print *
!    print *, "Second interaction"
!    call interaction_write (int_in2)
!    print *

    call find_connections (int_in1, int_in2, n_conn, connection_index)
    call compute_index_bounds_and_mappings &
         (int_in1, int_in2, n_conn, &
          n_in, n_vir, n_out, n_tot, &
          n_rest, prt_map_in, prt_map_conn)

    call prt_mask_init (prt_is_connected(1), interaction_get_n_tot (int_in1))
    call prt_mask_init (prt_is_connected(2), interaction_get_n_tot (int_in2))
    do i = 1, 2
       prt_is_connected(i)%entry = .true.
       prt_is_connected(i)%entry(connection_index(:,i)) = .false.
```

639

```
         end do
         allocate (qn_mask_conn_initial (n_conn))
         qn_mask_conn_initial = &
              interaction_get_mask (int_in1, connection_index(:,1)) .or. &
              interaction_get_mask (int_in2, connection_index(:,2))
         allocate (qn_mask_in(1)%mask (interaction_get_n_tot (int_in1)))
         allocate (qn_mask_in(2)%mask (interaction_get_n_tot (int_in2)))
         qn_mask_in(1)%mask = interaction_get_mask (int_in1)
         qn_mask_in(2)%mask = interaction_get_mask (int_in2)

         call connection_table_init (connection_table, &
              interaction_get_state_matrix (int_in1), &
              interaction_get_state_matrix (int_in2), &
              qn_mask_conn_initial,  &
              n_conn, connection_index, n_rest)
         call connection_table_fill (connection_table, &
              interaction_get_state_matrix (int_in1), &
              interaction_get_state_matrix (int_in2), &
              connection_index, prt_is_connected)
         call make_product_interaction (eval%int, &
              n_in, n_vir, n_out, &
              connection_table, &
              prt_map_in, prt_is_connected, &
              qn_mask_in, qn_mask_conn_initial, qn_mask_conn, qn_mask_rest)
!     call connection_table_write (connection_table)
         call make_pairing_array (eval%pairing_array, &
              interaction_get_n_matrix_elements (eval%int), &
              connection_table)
         call record_links (eval%int, &
              int_in1, int_in2, connection_index, prt_map_in, prt_map_conn, &
              prt_is_connected, connections_are_resonant)
         call connection_table_final (connection_table)

!     print *, "Result evaluator"
!     call evaluator_write (eval)

         if (interaction_get_n_matrix_elements (eval%int) == 0) then
            print *, "Evaluator product"
            print *, "First interaction"
            call interaction_write (int_in1)
            print *
            print *, "Second interaction"
            call interaction_write (int_in2)
            print *
            call msg_fatal ("Product of density matrices is empty", &
                       (/ var_str ("   ----------------------------------------"), &
                        var_str ("This happens when two density matrices are convoluted "), &
                        var_str ("but the processes they belong to (e.g., production "), &
                        var_str ("and decay) do not match.  This may indicate a WHIZARD bug.") /) )
         end if

      contains

      subroutine compute_index_bounds_and_mappings &
```

```fortran
    (int1, int2, n_conn, &
     n_in, n_vir, n_out, n_tot, &
     n_rest, prt_map_in, prt_map_conn)
  type(interaction_t), intent(in) :: int1, int2
  integer, intent(in) :: n_conn
  integer, intent(out) :: n_in, n_vir, n_out, n_tot
  integer, dimension(2), intent(out) :: n_rest
  type(index_map_t), dimension(2), intent(out) :: prt_map_in
  type(index_map_t), intent(out) :: prt_map_conn
  integer, dimension(:), allocatable :: index
  integer :: n_in1, n_vir1, n_out1
  integer :: n_in2, n_vir2, n_out2
  integer :: k
  n_in1  = interaction_get_n_in  (int1)
  n_vir1 = interaction_get_n_vir (int1)
  n_out1 = interaction_get_n_out (int1) - n_conn
  n_rest(1) = n_in1 + n_vir1 + n_out1
  n_in2  = interaction_get_n_in  (int2) - n_conn
  n_vir2 = interaction_get_n_vir (int2)
  n_out2 = interaction_get_n_out (int2)
  n_rest(2) = n_in2 + n_vir2 + n_out2
  n_in  = n_in1  + n_in2
  n_vir = n_vir1 + n_vir2 + n_conn
  n_out = n_out1 + n_out2
  n_tot = n_in + n_vir + n_out
  call index_map_init (prt_map_in, n_rest)
  call index_map_init (prt_map_conn, n_conn)
  allocate (index (n_tot))
  index = (/ (i, i = 1, n_tot) /)
  prt_map_in(1)%entry(1 : n_in1) = index(  1 :   n_in1)
  k =      n_in1
  prt_map_in(2)%entry(1 : n_in2) = index(k+1 : k+n_in2)
  k = k + n_in2
  prt_map_in(1)%entry(n_in1+1 : n_in1+n_vir1) = index(k+1 : k+n_vir1)
  k = k + n_vir1
  prt_map_in(2)%entry(n_in2+1 : n_in2+n_vir2) = index(k+1 : k+n_vir2)
  k = k + n_vir2
  prt_map_conn%entry = index(k+1 : k+n_conn)
  k = k + n_conn
  prt_map_in(1)%entry(n_in1+n_vir1+1 : n_rest(1)) = index(k+1 : k+n_out1)
  k = k + n_out1
  prt_map_in(2)%entry(n_in2+n_vir2+1 : n_rest(2)) = index(k+1 : k+n_out2)
end subroutine compute_index_bounds_and_mappings

subroutine connection_table_init &
    (connection_table, state_in1, state_in2, qn_mask_conn, &
     n_conn, connection_index, n_rest)
  type(connection_table_t), intent(out) :: connection_table
  type(state_matrix_t), intent(in), target :: state_in1, state_in2
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_conn
  integer, intent(in) :: n_conn
  integer, dimension(:,:), intent(in) :: connection_index
  integer, dimension(2), intent(in) :: n_rest
  integer, dimension(2) :: n_me_in
```

```fortran
    type(state_iterator_t) :: it
    type(quantum_numbers_t), dimension(n_conn) :: qn
    integer :: i, me_index_in, me_index_conn, n_me_conn
    integer, dimension(2) :: me_count
    connection_table%n_conn = n_conn
    connection_table%n_rest = n_rest
    n_me_in(1) = state_matrix_get_n_matrix_elements (state_in1)
    n_me_in(2) = state_matrix_get_n_matrix_elements (state_in2)
    allocate (connection_table%index_conn (2))
    call index_map_init (connection_table%index_conn, n_me_in)
    connection_table%index_conn = 0
    call state_matrix_init (connection_table%state, n_counters=2)
    do i = 1, 2
       select case (i)
       case (1);  call state_iterator_init (it, state_in1)
       case (2);  call state_iterator_init (it, state_in2)
       end select
       do while (state_iterator_is_valid (it))
          qn = state_iterator_get_quantum_numbers (it, connection_index(:,i))
          call quantum_numbers_undefine (qn, qn_mask_conn)
          call quantum_numbers_canonicalize_color (qn)
          me_index_in = state_iterator_get_me_index (it)
          call state_matrix_add_state (connection_table%state, qn, &
               counter_index = i, me_index = me_index_conn)
          call index_map_set_entry (connection_table%index_conn(i), &
               me_index_in, me_index_conn)
          call state_iterator_advance (it)
       end do
    end do
    n_me_conn = state_matrix_get_n_matrix_elements (connection_table%state)
    connection_table%n_me_conn = n_me_conn
    allocate (connection_table%entry (n_me_conn))
    call state_iterator_init (it, connection_table%state)
    do while (state_iterator_is_valid (it))
       i = state_iterator_get_me_index (it)
       me_count = state_iterator_get_me_count (it)
       call connection_entry_init (connection_table%entry(i), 2, 2, &
            state_iterator_get_quantum_numbers (it), &
            me_count, n_rest)
       call state_iterator_advance (it)
    end do
  end subroutine connection_table_init

  subroutine connection_table_final (connection_table)
    type(connection_table_t), intent(inout) :: connection_table
    call state_matrix_final (connection_table%state)
  end subroutine connection_table_final

  subroutine connection_table_write (connection_table, unit)
    type(connection_table_t), intent(in) :: connection_table
    integer, intent(in), optional :: unit
    integer :: i, j
    integer :: u
    u = output_unit (unit)
```

```
      write (u, *) "Connection table:"
      call state_matrix_write (connection_table%state, unit)
      if (allocated (connection_table%index_conn)) then
         write (u, *) "  Index mapping input => connection table:"
         do i = 1, size (connection_table%index_conn)
            write (u, *) "  Input state", i
            do j = 1, size (connection_table%index_conn(i))
               write (u, *)  j, &
                     index_map_get_entry (connection_table%index_conn(i), j)
            end do
         end do
      end if
      if (allocated (connection_table%entry)) then
         write (u, *) "  Connection table contents:"
         do i = 1, size (connection_table%entry)
            call connection_entry_write (connection_table%entry(i), unit)
         end do
      end if
      if (index_map_exists (connection_table%index_result)) then
         write (u, *) "  Index mapping connection table => output:"
         do i = 1, size (connection_table%index_result)
            write (u, *)  i, &
                  index_map_get_entry (connection_table%index_result, i)
         end do
      end if
    end subroutine connection_table_write

    subroutine connection_table_fill &
         (connection_table, state_in1, state_in2, &
          connection_index, prt_is_connected)
      type(connection_table_t), intent(inout) :: connection_table
      type(state_matrix_t), intent(in), target :: state_in1, state_in2
      integer, dimension(:,:), intent(in) :: connection_index
      type(prt_mask_t), dimension(2), intent(in) :: prt_is_connected
      type(state_iterator_t) :: it
      integer :: index_in, index_conn
      integer :: color_offset
      integer :: n_result_entries
      integer :: i, k
      color_offset = state_matrix_get_max_color_value (connection_table%state)
      do i = 1, 2
         select case (i)
         case (1);  call state_iterator_init (it, state_in1)
         case (2);  call state_iterator_init (it, state_in2)
         end select
         do while (state_iterator_is_valid (it))
            index_in = state_iterator_get_me_index (it)
            index_conn = index_map_get_entry &
                              (connection_table%index_conn(i), index_in)
            if (index_conn /= 0) then
               call connection_entry_add_state &
                     (connection_table%entry(index_conn), i, &
                      index_in, &
                      state_iterator_get_quantum_numbers (it), &
```

643

```
                    connection_index(:,i), prt_is_connected(i), &
                    color_offset)
          end if
          call state_iterator_advance (it)
       end do
       color_offset = color_offset &
            + state_matrix_get_max_color_value (state_in1)
    end do
    n_result_entries = 0
    do k = 1, size (connection_table%entry)
       n_result_entries = &
            n_result_entries + product (connection_table%entry(k)%n_index)
    end do
    call index_map_init (connection_table%index_result, n_result_entries)
  end subroutine connection_table_fill

  subroutine connection_entry_add_state &
       (entry, i, index_in, qn_in, connection_index, prt_is_connected, &
        color_offset)
    type(connection_entry_t), intent(inout) :: entry
    integer, intent(in) :: i
    integer, intent(in) :: index_in
    type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
    integer, dimension(:), intent(in) :: connection_index
    type(prt_mask_t), intent(in) :: prt_is_connected
    integer, intent(in) :: color_offset
    integer :: c, k
    integer, dimension(:,:), allocatable :: color_map
    entry%count(i) = entry%count(i) + 1
    c = entry%count(i)
    call quantum_numbers_set_color_map &
         (color_map, qn_in(connection_index), entry%qn_conn)
    call index_map_set_entry (entry%index_in(i), c, index_in)
    entry%qn_in_list(i)%qn(:,c) = pack (qn_in, prt_is_connected%entry)
    call quantum_numbers_translate_color &
         (entry%qn_in_list(i)%qn(:,c), color_map, color_offset)
  end subroutine connection_entry_add_state

  subroutine make_product_interaction (int, &
       n_in, n_vir, n_out, &
       connection_table, &
       prt_map_in, prt_is_connected, &
       qn_mask_in, qn_mask_conn_initial, qn_mask_conn, qn_mask_rest)
    type(interaction_t), intent(out), target :: int
    integer, intent(in) :: n_in, n_vir, n_out
    type(connection_table_t), intent(inout), target :: connection_table
    type(index_map_t), dimension(2), intent(in) :: prt_map_in
    type(prt_mask_t), dimension(2), intent(in) :: prt_is_connected
    type(qn_mask_array_t), dimension(2), intent(in) :: qn_mask_in
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: &
         qn_mask_conn_initial
    type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
    type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
    type(index_map_t), dimension(2) :: prt_index_in
```

```fortran
    type(index_map_t) :: prt_index_conn
    integer :: n_tot, n_conn
    integer, dimension(2) :: n_rest
    integer :: i, j, k, m
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
    type(connection_entry_t), pointer :: entry
    integer :: result_index
    n_conn = connection_table%n_conn
    n_rest = connection_table%n_rest
    n_tot = sum (n_rest) + n_conn
    allocate (qn (n_tot), qn_mask (n_tot))
    do i = 1, 2
       call index_map_init (prt_index_in(i), n_rest(i))
       prt_index_in(i) = &
            prt_map_in(i)%entry ((/ (j, j = 1, n_rest(i)) /))
    end do
    call index_map_init (prt_index_conn, n_conn)
    prt_index_conn = prt_map_conn%entry ((/ (j, j = 1, n_conn) /))
    do i = 1, 2
       if (present (qn_mask_rest)) then
          qn_mask(prt_index_in(i)%entry) = &
               pack (qn_mask_in(i)%mask, prt_is_connected(i)%entry) &
               .or. qn_mask_rest
       else
          qn_mask(prt_index_in(i)%entry) = &
               pack (qn_mask_in(i)%mask, prt_is_connected(i)%entry)
       end if
    end do
    qn_mask(prt_index_conn%entry) = qn_mask_conn_initial .or. qn_mask_conn
    call interaction_init (eval%int, n_in, n_vir, n_out, mask=qn_mask)
    m = 1
    do i = 1, connection_table%n_me_conn
       entry => connection_table%entry(i)
       qn(prt_index_conn%entry) = &
            quantum_numbers_undefined (entry%qn_conn, qn_mask_conn)
       do j = 1, entry%n_index(1)
          qn(prt_index_in(1)%entry) = entry%qn_in_list(1)%qn(:,j)
          do k = 1, entry%n_index(2)
             qn(prt_index_in(2)%entry) = entry%qn_in_list(2)%qn(:,k)
             call interaction_add_state (int, qn, me_index = result_index)
             call index_map_set_entry &
                  (connection_table%index_result, m, result_index)
             m = m + 1
          end do
       end do
    end do
    call interaction_freeze (int)
  end subroutine make_product_interaction

  subroutine make_pairing_array (pa, n_matrix_elements, connection_table)
    type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
    integer, intent(in) :: n_matrix_elements
    type(connection_table_t), intent(in), target :: connection_table
```

```
      type(connection_entry_t), pointer :: entry
      integer, dimension(:), allocatable :: n_entries
      integer :: i, j, k, m, r
      allocate (pa (n_matrix_elements))
      allocate (n_entries (n_matrix_elements))
      n_entries = 0
      do m = 1, size (connection_table%index_result)
         r = index_map_get_entry (connection_table%index_result, m)
         n_entries(r) = n_entries(r) + 1
      end do
      call pairing_array_init &
           (pa, n_entries, has_i2=.true., has_factor=.false.)
      m = 1
      n_entries = 0
      do i = 1, connection_table%n_me_conn
         entry => connection_table%entry(i)
         do j = 1, entry%n_index(1)
            do k = 1, entry%n_index(2)
               r = index_map_get_entry (connection_table%index_result, m)
               n_entries(r) = n_entries(r) + 1
               pa(r)%i1(n_entries(r)) = &
                    index_map_get_entry (entry%index_in(1), j)
               pa(r)%i2(n_entries(r)) = &
                    index_map_get_entry (entry%index_in(2), k)
               m = m + 1
            end do
         end do
      end do
    end subroutine make_pairing_array

    subroutine record_links (int, &
         int_in1, int_in2, connection_index, prt_map_in, prt_map_conn, &
         prt_is_connected, connections_are_resonant)
      type(interaction_t), intent(inout) :: int
      type(interaction_t), intent(in), target :: int_in1, int_in2
      integer, dimension(:,:), intent(in) :: connection_index
      type(index_map_t), dimension(2), intent(in) :: prt_map_in
      type(index_map_t), intent(in) :: prt_map_conn
      type(prt_mask_t), dimension(2), intent(in) :: prt_is_connected
      logical, intent(in), optional :: connections_are_resonant
      type(index_map_t), dimension(2) :: prt_map_all
      integer :: i, j, k, ival
      call index_map_init (prt_map_all(1), size (prt_is_connected(1)))
      k = 0
      j = 0
      do i = 1, size (prt_is_connected(1))
         if (prt_is_connected(1)%entry(i)) then
            j = j + 1
            ival = index_map_get_entry (prt_map_in(1), j)
            call index_map_set_entry (prt_map_all(1), i, ival)
         else
            k = k + 1
            ival = index_map_get_entry (prt_map_conn, k)
            call index_map_set_entry (prt_map_all(1), i, ival)
```

```
         end if
         call interaction_set_source_link (int, ival, int_in1, i)
      end do
      call interaction_transfer_relations (int_in1, int, prt_map_all(1)%entry)
      call index_map_init (prt_map_all(2), size (prt_is_connected(2)))
      j = 0
      do i = 1, size (prt_is_connected(2))
         if (prt_is_connected(2)%entry(i)) then
            j = j + 1
            ival = index_map_get_entry (prt_map_in(2), j)
            call index_map_set_entry (prt_map_all(2), i, ival)
            call interaction_set_source_link (int, ival, int_in2, i)
         else
            call index_map_set_entry (prt_map_all(2), i, 0)
         end if
      end do
      call interaction_transfer_relations (int_in2, int, prt_map_all(2)%entry)
      call interaction_relate_connections (int, &
           int_in2, connection_index(:,2), prt_map_all(2)%entry, &
           prt_map_conn%entry, connections_are_resonant)
    end subroutine record_links

  end subroutine evaluator_init_product_ii

  subroutine evaluator_init_product_ie &
       (eval, int_in1, eval_in2, qn_mask_conn, qn_mask_rest, &
        connections_are_resonant)
    type(evaluator_t), intent(out), target :: eval
    type(interaction_t), intent(in), target :: int_in1
    type(evaluator_t), intent(in), target :: eval_in2
    type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
    type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
    logical, intent(in), optional :: connections_are_resonant
    call evaluator_init_product_ii &
         (eval, int_in1, eval_in2%int, qn_mask_conn, qn_mask_rest, &
          connections_are_resonant)
  end subroutine evaluator_init_product_ie

  subroutine evaluator_init_product_ei &
       (eval, eval_in1, int_in2, qn_mask_conn, qn_mask_rest, &
        connections_are_resonant)
    type(evaluator_t), intent(out), target :: eval
    type(evaluator_t), intent(in), target :: eval_in1
    type(interaction_t), intent(in), target :: int_in2
    type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
    type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
    logical, intent(in), optional :: connections_are_resonant
    call evaluator_init_product_ii &
         (eval, eval_in1%int, int_in2, qn_mask_conn, qn_mask_rest, &
          connections_are_resonant)
  end subroutine evaluator_init_product_ei

  subroutine evaluator_init_product_ee &
       (eval, eval_in1, eval_in2, qn_mask_conn, qn_mask_rest, &
```

```
        connections_are_resonant)
    type(evaluator_t), intent(out), target :: eval
    type(evaluator_t), intent(in), target :: eval_in1, eval_in2
    type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
    type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
    logical, intent(in), optional :: connections_are_resonant
    call evaluator_init_product_ii &
         (eval, eval_in1%int, eval_in2%int, qn_mask_conn, qn_mask_rest, &
          connections_are_resonant)
  end subroutine evaluator_init_product_ee
```

7.7.5 Creating an evaluator: square

The generic initializer for an evaluator that squares a matrix element. Depending on the provided mask, we select the appropriate specific initializer for either diagonal or non-diagonal helicity density matrices.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_init_square
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_init_square (eval, int_in, qn_mask, &
         col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)
      type(evaluator_t), intent(out), target :: eval
      type(interaction_t), intent(in), target :: int_in
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
      integer, dimension(:,:), intent(in), optional :: col_flow_index
      complex(default), dimension(:), intent(in), optional :: col_factor
      integer, dimension(:), intent(in), optional :: col_index_hi
      logical, intent(in), optional :: expand_color_flows
      integer, intent(in), optional :: nc
      if (all (quantum_numbers_mask_diagonal_helicity (qn_mask))) then
         call evaluator_init_square_diag (eval, int_in, qn_mask, &
              col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)
      else
         call evaluator_init_square_nondiag (eval, int_in, qn_mask, &
              col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)
      end if
    end subroutine evaluator_init_square
```

Color-summed squared matrix (diagonal helicities)

The initializer for an evaluator that squares a matrix element, including color factors. The mask must be such that off-diagonal matrix elements are excluded.

If `color_flows` is set, the evaluator keeps color-flow entries separate and drops all interfering color structures. The color factors are set to unity in this case.

There is only one input interaction. The quantum-number mask is an array, one entry for each particle, so they can be treated individually. For academic purposes, we allow for the number of colors being different from three (but 3 is the default).

The algorithm is analogous to multiplication, with a few notable differences:

1. The connected particles are known, the correspondence is one-to-one. All particles are connected, and the mapping of indices is trivial, which simplifies the following steps.

2. `accumulate_connected_states`: The matrix of connected states encompasses all particles, but color indices are removed. However, ghost states are still kept separate from physical color states. No color-index reassignment is necessary.

3. The table of connections contains single index and quantum-number arrays instead of pairs of them. They are paired with themselves in all possible ways.

4. `make_squared_interaction`: Now apply the predefined quantum-numbers mask, which usually collects all color states (physical and ghosts), and possibly a helicity sum.

5. `make_pairing_array`: For each pair of input states, compute the color factor (including a potential ghost-parity sign) and store this in the pairing array together with the matrix-element indices for multiplication.

6. `record_links`: This is again trivial due to the one-to-one correspondence.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_init_square_diag
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_init_square_diag (eval, int_in, qn_mask, &
        col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)

      type(evaluator_t), intent(out), target :: eval
      type(interaction_t), intent(in), target :: int_in
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
      integer, dimension(:,:), intent(in), optional :: col_flow_index
      complex(default), dimension(:), intent(in), optional :: col_factor
      integer, dimension(:), intent(in), optional :: col_index_hi
      logical, intent(in), optional :: expand_color_flows
      integer, intent(in), optional :: nc

      integer :: n_in, n_vir, n_out, n_tot
      integer :: n_me_in
      type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask_initial

      type :: connection_table_t
        integer :: n_tot = 0
        integer :: n_me_conn = 0
        type(state_matrix_t) :: state
        type(index_map_t) :: index_conn
        type(connection_entry_t), dimension(:), allocatable :: entry
        type(index_map_t) :: index_result
      end type connection_table_t
      type(connection_table_t) :: connection_table

      logical :: sum_colors
      type(color_table_t) :: color_table
```

```
      if (present (expand_color_flows)) then
         sum_colors = .not. expand_color_flows
      else
         sum_colors = .true.
      end if

      if (sum_colors) then
         eval%type = EVAL_SQUARE_WITH_COLOR_FACTORS
      else
         eval%type = EVAL_SQUARED_FLOWS
      end if
      eval%int_in1 => int_in

!     print *, "Interaction square with color factors (diag)"
!     print *, "Input interaction"
!     call interaction_write (int_in)

      n_in  = interaction_get_n_in  (int_in)
      n_vir = interaction_get_n_vir (int_in)
      n_out = interaction_get_n_out (int_in)
      n_tot = interaction_get_n_tot (int_in)

      allocate (qn_mask_initial (n_tot))
      qn_mask_initial = interaction_get_mask (int_in)
      call quantum_numbers_mask_set_color &
           (qn_mask_initial, sum_colors, mask_cg=.false.)
      if (sum_colors) then
         call color_table_init &
              (color_table, interaction_get_state_matrix (int_in), n_tot)
         if (present (col_flow_index) .and. present (col_factor) &
              .and. present (col_index_hi)) then
            call color_table_set_color_factors &
                 (color_table, col_flow_index, col_factor, col_index_hi)
         end if
!        call color_table_write (color_table)
      end if

      call connection_table_init (connection_table, &
           interaction_get_state_matrix (int_in), qn_mask_initial, qn_mask, n_tot)
      call connection_table_fill (connection_table, &
           interaction_get_state_matrix (int_in))
      call make_squared_interaction (eval%int, &
           n_in, n_vir, n_out, n_tot, &
           connection_table, sum_colors, qn_mask_initial .or. qn_mask)
      call make_pairing_array (eval%pairing_array, &
           interaction_get_n_matrix_elements (eval%int), &
           connection_table, sum_colors, color_table, n_in, n_tot, nc)
      call record_links (eval%int, int_in, n_tot)
      call connection_table_final (connection_table)
!     print *, "Result evaluator:"
!     call evaluator_write (eval)

    contains
```

650

```
subroutine connection_table_init &
     (connection_table, state_in, qn_mask_in, qn_mask, n_tot)
  type(connection_table_t), intent(out) :: connection_table
  type(state_matrix_t), intent(in), target :: state_in
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
  integer, intent(in) :: n_tot
  type(quantum_numbers_t), dimension(n_tot) :: qn
  type(state_iterator_t) :: it
  integer :: i, n_me_in, me_index_in
  integer :: me_index_conn, n_me_conn
  integer, dimension(1) :: me_count
  connection_table%n_tot = n_tot
  n_me_in = state_matrix_get_n_matrix_elements (state_in)
  call index_map_init (connection_table%index_conn, n_me_in)
  connection_table%index_conn = 0
  call state_matrix_init (connection_table%state, n_counters=1)
  call state_iterator_init (it, state_in)
  do while (state_iterator_is_valid (it))
     qn = state_iterator_get_quantum_numbers (it)
     if (all (quantum_numbers_are_physical (qn, qn_mask))) then
        call quantum_numbers_undefine (qn, qn_mask_in)
        me_index_in = state_iterator_get_me_index (it)
        call state_matrix_add_state (connection_table%state, qn, &
             counter_index = 1, me_index = me_index_conn)
        call index_map_set_entry (connection_table%index_conn, &
             me_index_in, me_index_conn)
     end if
     call state_iterator_advance (it)
  end do
  n_me_conn = state_matrix_get_n_matrix_elements (connection_table%state)
  connection_table%n_me_conn = n_me_conn
  allocate (connection_table%entry (n_me_conn))
  call state_iterator_init (it, connection_table%state)
  do while (state_iterator_is_valid (it))
     i = state_iterator_get_me_index (it)
     me_count = state_iterator_get_me_count (it)
     call connection_entry_init (connection_table%entry(i), 1, 2, &
          state_iterator_get_quantum_numbers (it), me_count, (/n_tot/))
     call state_iterator_advance (it)
  end do
end subroutine connection_table_init

subroutine connection_table_final (connection_table)
  type(connection_table_t), intent(inout) :: connection_table
  call state_matrix_final (connection_table%state)
end subroutine connection_table_final

subroutine connection_table_write (connection_table, unit)
  type(connection_table_t), intent(in) :: connection_table
  integer, intent(in), optional :: unit
  integer :: i, j
  integer :: u
```

```
   u = output_unit (unit)
   write (u, *) "Connection table:"
   call state_matrix_write (connection_table%state, unit)
   if (index_map_exists (connection_table%index_conn)) then
      write (u, *) "  Index mapping input => connection table:"
      do i = 1, size (connection_table%index_conn)
         write (u, *)  i, &
                 index_map_get_entry (connection_table%index_conn, i)
      end do
   end if
   if (allocated (connection_table%entry)) then
      write (u, *) "  Connection table contents"
      do i = 1, size (connection_table%entry)
         call connection_entry_write (connection_table%entry(i), unit)
      end do
   end if
   if (index_map_exists (connection_table%index_result)) then
      write (u, *) "  Index mapping connection table => output"
      do i = 1, size (connection_table%index_result)
         write (u, *)  i, &
                 index_map_get_entry (connection_table%index_result, i)
      end do
   end if
end subroutine connection_table_write

subroutine connection_table_fill (connection_table, state)
  type(connection_table_t), intent(inout) :: connection_table
  type(state_matrix_t), intent(in), target :: state
  integer :: index_in, index_conn, n_result_entries
  type(state_iterator_t) :: it
  integer :: k
  call state_iterator_init (it, state)
  do while (state_iterator_is_valid (it))
     index_in = state_iterator_get_me_index (it)
     index_conn = &
          index_map_get_entry (connection_table%index_conn, index_in)
     if (index_conn /= 0) then
        call connection_entry_add_state &
             (connection_table%entry(index_conn), &
              index_in, &
              state_iterator_get_quantum_numbers (it))
     end if
     call state_iterator_advance (it)
  end do
  n_result_entries = 0
  do k = 1, size (connection_table%entry)
     n_result_entries = &
          n_result_entries + connection_table%entry(k)%n_index(1) ** 2
  end do
  call index_map_init (connection_table%index_result, n_result_entries)
  connection_table%index_result = 0
end subroutine connection_table_fill

subroutine connection_entry_add_state (entry, index_in, qn_in)
```

```fortran
    type(connection_entry_t), intent(inout) :: entry
    integer, intent(in) :: index_in
    type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
    integer :: c
    entry%count = entry%count + 1
    c = entry%count(1)
    call index_map_set_entry (entry%index_in(1), c, index_in)
    entry%qn_in_list(1)%qn(:,c) = qn_in
  end subroutine connection_entry_add_state

  subroutine make_squared_interaction (int, &
       n_in, n_vir, n_out, n_tot, &
       connection_table, sum_colors, qn_mask)
    type(interaction_t), intent(out), target :: int
    integer, intent(in) :: n_in, n_vir, n_out, n_tot
    type(connection_table_t), intent(inout), target :: connection_table
    logical, intent(in) :: sum_colors
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
    type(connection_entry_t), pointer :: entry
    integer :: result_index, n_contrib
    integer :: i, m
    type(quantum_numbers_t), dimension(n_tot) :: qn
    call interaction_init (eval%int, n_in, n_vir, n_out, mask=qn_mask)
    m = 0
    do i = 1, connection_table%n_me_conn
       entry => connection_table%entry(i)
       qn = quantum_numbers_undefined (entry%qn_conn, qn_mask)
       if (.not. sum_colors)   call quantum_numbers_invert_color (qn(1:n_in))
       call interaction_add_state (int, qn, me_index = result_index)
       n_contrib = entry%n_index(1) ** 2
       connection_table%index_result%entry(m+1:m+n_contrib) = result_index
       m = m + n_contrib
    end do
    call interaction_freeze (int)
  end subroutine make_squared_interaction

  subroutine make_pairing_array (pa, &
       n_matrix_elements, connection_table, sum_colors, color_table, &
       n_in, n_tot, nc)
    type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
    integer, intent(in) :: n_matrix_elements
    type(connection_table_t), intent(in), target :: connection_table
    logical, intent(in) :: sum_colors
    type(color_table_t), intent(inout) :: color_table
    type(connection_entry_t), pointer :: entry
    integer, intent(in) :: n_in, n_tot
    integer, intent(in), optional :: nc
    integer, dimension(:), allocatable :: n_entries
    integer :: i, j, k, l, ks, ls, m, r
    integer :: color_multiplicity_in
    allocate (pa (n_matrix_elements))
    allocate (n_entries (n_matrix_elements))
    n_entries = 0
    do m = 1, size (connection_table%index_result)
```

```fortran
            r = index_map_get_entry (connection_table%index_result, m)
            n_entries(r) = n_entries(r) + 1
         end do
      call pairing_array_init &
           (pa, n_entries, has_i2 = sum_colors, has_factor = sum_colors)
      m = 1
      n_entries = 0
      do i = 1, connection_table%n_me_conn
         entry => connection_table%entry(i)
         do k = 1, entry%n_index(1)
            if (sum_colors) then
               color_multiplicity_in = &
                    product (abs (quantum_numbers_get_color_type &
                                   (entry%qn_in_list(1)%qn(:n_in, k))))
               do l = 1, entry%n_index(1)
                  r = index_map_get_entry (connection_table%index_result, m)
                  n_entries(r) = n_entries(r) + 1
                  ks = index_map_get_entry (entry%index_in(1), k)
                  ls = index_map_get_entry (entry%index_in(1), l)
                  pa(r)%i1(n_entries(r)) = ks
                  pa(r)%i2(n_entries(r)) = ls
                  pa(r)%factor(n_entries(r)) = &
                       color_table_get_color_factor (color_table, ks, ls, nc) &
                       / color_multiplicity_in
                  m = m + 1
               end do
            else
               r = index_map_get_entry (connection_table%index_result, m)
               n_entries(r) = n_entries(r) + 1
               ks = index_map_get_entry (entry%index_in(1), k)
               pa(r)%i1(n_entries(r)) = ks
               m = m + 1
            end if
         end do
      end do
    end subroutine make_pairing_array

    subroutine record_links (int, int_in, n_tot)
      type(interaction_t), intent(inout) :: int
      type(interaction_t), intent(in), target :: int_in
      integer, intent(in) :: n_tot
      integer, dimension(n_tot) :: map
      integer :: i
      do i = 1, n_tot
         call interaction_set_source_link (int, i, int_in, i)
      end do
      map = (/ (i, i = 1, n_tot) /)
      call interaction_transfer_relations (int_in, int, map)
    end subroutine record_links

  end subroutine evaluator_init_square_diag
```

Color-summed squared matrix (support nodiagonal helicities)

The initializer for an evaluator that squares a matrix element, including color factors. Unless requested otherwise by the quantum-number mask, the result contains off-diagonal matrix elements. (The input interaction must be diagonal since it represents an amplitude, not a density matrix.)

There is only one input interaction. The quantum-number mask is an array, one entry for each particle, so they can be treated individually. For academic purposes, we allow for the number of colors being different from three (but 3 is the default).

The algorithm is analogous to the previous one, with some additional complications due to the necessity to loop over two helicity indices.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_init_square_nondiag
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_init_square_nondiag (eval, int_in, qn_mask, &
        col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)

      type(evaluator_t), intent(out), target :: eval
      type(interaction_t), intent(in), target :: int_in
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
      integer, dimension(:,:), intent(in), optional :: col_flow_index
      complex(default), dimension(:), intent(in), optional :: col_factor
      integer, dimension(:), intent(in), optional :: col_index_hi
      logical, intent(in), optional :: expand_color_flows
      integer, intent(in), optional :: nc

      integer :: n_in, n_vir, n_out, n_tot
      integer :: n_me_in
      type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask_initial

      type :: connection_table_t
        integer :: n_tot = 0
        integer :: n_me_conn = 0
        type(state_matrix_t) :: state
        type(index_map2_t) :: index_conn
        type(connection_entry_t), dimension(:), allocatable :: entry
        type(index_map_t) :: index_result
      end type connection_table_t
      type(connection_table_t) :: connection_table

      logical :: sum_colors
      type(color_table_t) :: color_table

      if (present (expand_color_flows)) then
         sum_colors = .not. expand_color_flows
      else
         sum_colors = .true.
      end if

      if (sum_colors) then
         eval%type = EVAL_SQUARE_WITH_COLOR_FACTORS
      else
```

```
      eval%type = EVAL_SQUARED_FLOWS
    end if
    eval%int_in1 => int_in

!    print *, "Interaction square with color factors (nondiag)"
!    print *, "Input interaction"
!    call interaction_write (int_in)
    n_in  = interaction_get_n_in  (int_in)
    n_vir = interaction_get_n_vir (int_in)
    n_out = interaction_get_n_out (int_in)
    n_tot = interaction_get_n_tot (int_in)

    allocate (qn_mask_initial (n_tot))
    qn_mask_initial = interaction_get_mask (int_in)
    call quantum_numbers_mask_set_color &
         (qn_mask_initial, sum_colors, mask_cg=.false.)
    if (sum_colors) then
       call color_table_init &
            (color_table, interaction_get_state_matrix (int_in), n_tot)
       if (present (col_flow_index) .and. present (col_factor) &
           .and. present (col_index_hi)) then
         call color_table_set_color_factors &
              (color_table, col_flow_index, col_factor, col_index_hi)
       end if
!       call color_table_write (color_table)
    end if

    call connection_table_init (connection_table, &
         interaction_get_state_matrix (int_in), qn_mask_initial, qn_mask, n_tot)
    call connection_table_fill (connection_table, &
         interaction_get_state_matrix (int_in))
    call make_squared_interaction (eval%int, &
         n_in, n_vir, n_out, n_tot, &
         connection_table, sum_colors, qn_mask_initial .or. qn_mask)
!    call connection_table_write (connection_table)
    call make_pairing_array (eval%pairing_array, &
         interaction_get_n_matrix_elements (eval%int), &
         connection_table, sum_colors, color_table, n_in, n_tot, nc)
    call record_links (eval%int, int_in, n_tot)
    call connection_table_final (connection_table)

!    print *, "Result evaluator:"
!    call evaluator_write (eval)

  contains

    subroutine connection_table_init &
         (connection_table, state_in, qn_mask_in, qn_mask, n_tot)
      type(connection_table_t), intent(out) :: connection_table
      type(state_matrix_t), intent(in), target :: state_in
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
      integer, intent(in) :: n_tot
      type(quantum_numbers_t), dimension(n_tot) :: qn1, qn2, qn
```

```
    type(state_iterator_t) :: it1, it2, it
    integer :: i, n_me_in, me_index_in1, me_index_in2
    integer :: me_index_conn, n_me_conn
    integer, dimension(1) :: me_count
    connection_table%n_tot = n_tot
    n_me_in = state_matrix_get_n_matrix_elements (state_in)
    call index_map2_init (connection_table%index_conn, n_me_in)
    connection_table%index_conn = 0
    call state_matrix_init (connection_table%state, n_counters=1)
    call state_iterator_init (it1, state_in)
    do while (state_iterator_is_valid (it1))
       qn1 = state_iterator_get_quantum_numbers (it1)
       me_index_in1 = state_iterator_get_me_index (it1)
       call state_iterator_init (it2, state_in)
       do while (state_iterator_is_valid (it2))
          qn2 = state_iterator_get_quantum_numbers (it2)
          if (all (quantum_numbers_are_compatible (qn1, qn2, qn_mask))) then
             qn = qn1 .merge. qn2
             call quantum_numbers_undefine (qn, qn_mask_in)
             me_index_in2 = state_iterator_get_me_index (it2)
             call state_matrix_add_state (connection_table%state, qn, &
                  counter_index = 1, me_index = me_index_conn)
             call index_map2_set_entry (connection_table%index_conn, &
                  me_index_in1, me_index_in2, me_index_conn)
          end if
          call state_iterator_advance (it2)
       end do
       call state_iterator_advance (it1)
    end do
    n_me_conn = state_matrix_get_n_matrix_elements (connection_table%state)
    connection_table%n_me_conn = n_me_conn
    allocate (connection_table%entry (n_me_conn))
    call state_iterator_init (it, connection_table%state)
    do while (state_iterator_is_valid (it))
       i = state_iterator_get_me_index (it)
       me_count = state_iterator_get_me_count (it)
       call connection_entry_init (connection_table%entry(i), 1, 2, &
            state_iterator_get_quantum_numbers (it), me_count, (/n_tot/))
       call state_iterator_advance (it)
    end do
  end subroutine connection_table_init

  subroutine connection_table_final (connection_table)
    type(connection_table_t), intent(inout) :: connection_table
    call state_matrix_final (connection_table%state)
  end subroutine connection_table_final

  subroutine connection_table_write (connection_table, unit)
    type(connection_table_t), intent(in) :: connection_table
    integer, intent(in), optional :: unit
    integer :: i, j
    integer :: u
    u = output_unit (unit)
    write (u, *) "Connection table:"
```

```
      call state_matrix_write (connection_table%state, unit)
      if (index_map2_exists (connection_table%index_conn)) then
         write (u, *) "  Index mapping input => connection table:"
         do i = 1, size (connection_table%index_conn)
            do j = 1, size (connection_table%index_conn)
               write (u, *)  i, j, &
                    index_map2_get_entry (connection_table%index_conn, i, j)
            end do
         end do
      end if
      if (allocated (connection_table%entry)) then
         write (u, *) "  Connection table contents"
         do i = 1, size (connection_table%entry)
            call connection_entry_write (connection_table%entry(i), unit)
         end do
      end if
      if (index_map_exists (connection_table%index_result)) then
         write (u, *) "  Index mapping connection table => output"
         do i = 1, size (connection_table%index_result)
            write (u, *)  i, &
                 index_map_get_entry (connection_table%index_result, i)
         end do
      end if
    end subroutine connection_table_write

    subroutine connection_table_fill (connection_table, state)
      type(connection_table_t), intent(inout), target :: connection_table
      type(state_matrix_t), intent(in), target :: state
      integer :: index1_in, index2_in, index_conn, n_result_entries
      type(state_iterator_t) :: it1, it2
      integer :: k
      call state_iterator_init (it1, state)
      do while (state_iterator_is_valid (it1))
         index1_in = state_iterator_get_me_index (it1)
         call state_iterator_init (it2, state)
         do while (state_iterator_is_valid (it2))
            index2_in = state_iterator_get_me_index (it2)
            index_conn = index_map2_get_entry &
                          (connection_table%index_conn, index1_in, index2_in)
            if (index_conn /= 0) then
               call connection_entry_add_state &
                    (connection_table%entry(index_conn), &
                     index1_in, index2_in, &
                     state_iterator_get_quantum_numbers (it1) &
                     .merge. &
                     state_iterator_get_quantum_numbers (it2))
            end if
            call state_iterator_advance (it2)
         end do
         call state_iterator_advance (it1)
      end do
      n_result_entries = 0
      do k = 1, size (connection_table%entry)
         n_result_entries = &
```

```
              n_result_entries + connection_table%entry(k)%n_index(1)
     end do
     call index_map_init (connection_table%index_result, n_result_entries)
     connection_table%index_result = 0
  end subroutine connection_table_fill

  subroutine connection_entry_add_state (entry, index1_in, index2_in, qn_in)
    type(connection_entry_t), intent(inout) :: entry
    integer, intent(in) :: index1_in, index2_in
    type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
    integer :: c
    entry%count = entry%count + 1
    c = entry%count(1)
    call index_map_set_entry (entry%index_in(1), c, index1_in)
    call index_map_set_entry (entry%index_in(2), c, index2_in)
    entry%qn_in_list(1)%qn(:,c) = qn_in
  end subroutine connection_entry_add_state

  subroutine make_squared_interaction (int, &
       n_in, n_vir, n_out, n_tot, &
       connection_table, sum_colors, qn_mask)
    type(interaction_t), intent(out), target :: int
    integer, intent(in) :: n_in, n_vir, n_out, n_tot
    type(connection_table_t), intent(inout), target :: connection_table
    logical, intent(in) :: sum_colors
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
    type(connection_entry_t), pointer :: entry
    integer :: result_index
    integer :: i, k, m
    type(quantum_numbers_t), dimension(n_tot) :: qn
    call interaction_init (eval%int, n_in, n_vir, n_out, mask=qn_mask)
    m = 0
    do i = 1, connection_table%n_me_conn
       entry => connection_table%entry(i)
       do k = 1, size (entry%qn_in_list(1)%qn, 2)
          qn = quantum_numbers_undefined &
                 (entry%qn_in_list(1)%qn(:,k), qn_mask)
          if (.not. sum_colors) &
                 call quantum_numbers_invert_color (qn(1:n_in))
          call interaction_add_state (int, qn, me_index = result_index)
          call index_map_set_entry (connection_table%index_result, m + 1, &
               result_index)
          m = m + 1
       end do
    end do
    call interaction_freeze (int)
  end subroutine make_squared_interaction

  subroutine make_pairing_array (pa, &
       n_matrix_elements, connection_table, sum_colors, color_table, &
       n_in, n_tot, nc)
    type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
    integer, intent(in) :: n_matrix_elements
    type(connection_table_t), intent(in), target :: connection_table
```

```fortran
    logical, intent(in) :: sum_colors
    type(color_table_t), intent(inout) :: color_table
    type(connection_entry_t), pointer :: entry
    integer, intent(in) :: n_in, n_tot
    integer, intent(in), optional :: nc
    integer, dimension(:), allocatable :: n_entries
    integer :: i, k, k1s, k2s, m, r
    integer :: color_multiplicity_in
    allocate (pa (n_matrix_elements))
    allocate (n_entries (n_matrix_elements))
    n_entries = 0
    do m = 1, size (connection_table%index_result)
       r = index_map_get_entry (connection_table%index_result, m)
       n_entries(r) = n_entries(r) + 1
    end do
    call pairing_array_init &
         (pa, n_entries, has_i2 = sum_colors, has_factor = sum_colors)
    m = 1
    n_entries = 0
    do i = 1, connection_table%n_me_conn
       entry => connection_table%entry(i)
       do k = 1, entry%n_index(1)
          r = index_map_get_entry (connection_table%index_result, m)
          n_entries(r) = n_entries(r) + 1
          if (sum_colors) then
             k1s = index_map_get_entry (entry%index_in(1), k)
             k2s = index_map_get_entry (entry%index_in(2), k)
             pa(r)%i1(n_entries(r)) = k1s
             pa(r)%i2(n_entries(r)) = k2s
             color_multiplicity_in = &
                   product (abs (quantum_numbers_get_color_type &
                                (entry%qn_in_list(1)%qn(:n_in, k))))
             pa(r)%factor(n_entries(r)) = &
                   color_table_get_color_factor (color_table, k1s, k2s, nc) &
                   / color_multiplicity_in
          else
             k1s = index_map_get_entry (entry%index_in(1), k)
             pa(r)%i1(n_entries(r)) = k1s
          end if
          m = m + 1
       end do
    end do
  end subroutine make_pairing_array

  subroutine record_links (int, int_in, n_tot)
    type(interaction_t), intent(inout) :: int
    type(interaction_t), intent(in), target :: int_in
    integer, intent(in) :: n_tot
    integer, dimension(n_tot) :: map
    integer :: i
    do i = 1, n_tot
       call interaction_set_source_link (int, i, int_in, i)
    end do
    map = (/ (i, i = 1, n_tot) /)
```

```
      call interaction_transfer_relations (int_in, int, map)
    end subroutine record_links

  end subroutine evaluator_init_square_nondiag
```

Copy with additional contracted color states

This evaluator involves no square or multiplication, its matrix elements are
just copies of the (single) input interaction. However, the state matrix of the
interaction contains additional states that have color indices contracted. This
is used for copies of the beam or structure-function interactions that need to
match the hard interaction also in the case where its color indices coincide.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_init_color_contractions
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_init_color_contractions (eval, int_in)
      type(evaluator_t), intent(out), target :: eval
      type(interaction_t), intent(in), target :: int_in
      integer :: n_in, n_vir, n_out, n_tot
      type(state_matrix_t), target :: state_with_contractions
      integer, dimension(:), allocatable :: me_index
      integer, dimension(:), allocatable :: result_index
      eval%type = EVAL_COLOR_CONTRACTION
      eval%int_in1 => int_in
 !    print *, "Interaction with additional color contractions"
 !    print *, "Input interaction"
 !    call interaction_write (int_in)
      n_in  = interaction_get_n_in  (int_in)
      n_vir = interaction_get_n_vir (int_in)
      n_out = interaction_get_n_out (int_in)
      n_tot = interaction_get_n_tot (int_in)
      state_with_contractions = interaction_get_state_matrix (int_in)
      call state_matrix_add_color_contractions (state_with_contractions)
      call make_contracted_interaction (eval%int, &
           me_index, result_index, &
           n_in, n_vir, n_out, n_tot, &
           state_with_contractions, interaction_get_mask (int_in))
      call make_pairing_array (eval%pairing_array, me_index, result_index)
      call record_links (eval%int, int_in, n_tot)
      call state_matrix_final (state_with_contractions)
 !    print *, "Result evaluator:"
 !    call evaluator_write (eval)

    contains

      subroutine make_contracted_interaction (int, &
           me_index, result_index, &
           n_in, n_vir, n_out, n_tot, state, qn_mask)
        type(interaction_t), intent(out), target :: int
        integer, dimension(:), intent(out), allocatable :: me_index
        integer, dimension(:), intent(out), allocatable :: result_index
        integer, intent(in) :: n_in, n_vir, n_out, n_tot
```

661

```fortran
    type(state_matrix_t), intent(in) :: state
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
    type(state_iterator_t) :: it
    integer :: n_me, i
    type(quantum_numbers_t), dimension(n_tot) :: qn
    call interaction_init (int, n_in, n_vir, n_out, mask=qn_mask)
    n_me = state_matrix_get_n_leaves (state)
    allocate (me_index (n_me))
    allocate (result_index (n_me))
    call state_iterator_init (it, state)
    i = 0
    do while (state_iterator_is_valid (it))
       i = i + 1
       me_index(i) = state_iterator_get_me_index (it)
       qn = state_iterator_get_quantum_numbers (it)
       call interaction_add_state (int, qn, me_index = result_index(i))
       call state_iterator_advance (it)
    end do
    call interaction_freeze (int)
  end subroutine make_contracted_interaction

  subroutine make_pairing_array (pa, me_index, result_index)
    type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
    integer, dimension(:), intent(in) :: me_index, result_index
    integer, dimension(:), allocatable :: n_entries
    integer :: n_matrix_elements, r, i
    n_matrix_elements = size (me_index)
    allocate (pa (n_matrix_elements))
    allocate (n_entries (n_matrix_elements))
    n_entries = 1
    call pairing_array_init &
         (pa, n_entries, has_i2=.false., has_factor=.false.)
    do i = 1, n_matrix_elements
       r = result_index(i)
       pa(r)%i1(1) = me_index(i)
    end do
  end subroutine make_pairing_array

  subroutine record_links (int, int_in, n_tot)
    type(interaction_t), intent(inout) :: int
    type(interaction_t), intent(in), target :: int_in
    integer, intent(in) :: n_tot
    integer, dimension(n_tot) :: map
    integer :: i
    do i = 1, n_tot
       call interaction_set_source_link (int, i, int_in, i)
    end do
    map = (/ (i, i = 1, n_tot) /)
    call interaction_transfer_relations (int_in, int, map)
  end subroutine record_links

end subroutine evaluator_init_color_contractions
```

Auxiliary procedure for initialization

This will become a standard procedure in F2008. The result is true if the number of true values in the mask is odd. We use the function for determining the ghost parity of a quantum-number array.

⟨*Evaluators: procedures*⟩+≡
```
function parity (mask)
  logical :: parity
  logical, dimension(:) :: mask
  integer :: i
  parity = .false.
  do i = 1, size (mask)
     if (mask(i))  parity = .not. parity
  end do
end function parity
```

7.7.6   Accessing contents

Return the interaction component, as a pointer to avoid any copying.

⟨*Evaluators: public*⟩+≡
```
public :: evaluator_get_int_ptr
```

⟨*Evaluators: procedures*⟩+≡
```
function evaluator_get_int_ptr (eval) result (int)
  type(interaction_t), pointer :: int
  type(evaluator_t), intent(in), target :: eval
  int => eval%int
end function evaluator_get_int_ptr
```

7.7.7   Inherited procedures

Return true if the state matrix within the interaction is empty.

⟨*Evaluators: public*⟩+≡
```
public :: evaluator_is_empty
```

⟨*Evaluators: procedures*⟩+≡
```
function evaluator_is_empty (eval) result (flag)
  logical :: flag
  type(evaluator_t), intent(in) :: eval
  flag = interaction_is_empty (eval%int)
end function evaluator_is_empty
```

Return the integer tag.

⟨*Evaluators: public*⟩+≡
```
public :: evaluator_get_tag
```

⟨*Evaluators: procedures*⟩+≡
```
function evaluator_get_tag (eval) result (tag)
  integer :: tag
  type(evaluator_t), intent(in) :: eval
  tag = interaction_get_tag (eval%int)
```

```
      end function evaluator_get_tag
```

Get the norm of the state matrix (if the norm has been taken out, otherwise
this would be unity).

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_get_norm
```

⟨*Evaluators: procedures*⟩+≡
```
    function evaluator_get_norm (eval) result (norm)
      real(default) :: norm
      type(evaluator_t), intent(in) :: eval
      norm = interaction_get_norm (eval%int)
    end function evaluator_get_norm
```

Return the number of particles.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_get_n_tot
    public :: evaluator_get_n_in
    public :: evaluator_get_n_vir
    public :: evaluator_get_n_out
```

⟨*Evaluators: procedures*⟩+≡
```
    function evaluator_get_n_tot (eval) result (n_tot)
      integer :: n_tot
      type(evaluator_t), intent(in) :: eval
      n_tot = interaction_get_n_tot (eval%int)
    end function evaluator_get_n_tot

    function evaluator_get_n_in (eval) result (n_in)
      integer :: n_in
      type(evaluator_t), intent(in) :: eval
      n_in = interaction_get_n_in (eval%int)
    end function evaluator_get_n_in

    function evaluator_get_n_vir (eval) result (n_vir)
      integer :: n_vir
      type(evaluator_t), intent(in) :: eval
      n_vir = interaction_get_n_vir (eval%int)
    end function evaluator_get_n_vir

    function evaluator_get_n_out (eval) result (n_out)
      integer :: n_out
      type(evaluator_t), intent(in) :: eval
      n_out = interaction_get_n_out (eval%int)
    end function evaluator_get_n_out
```

Sum all matrix element values.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_sum
```

⟨*Evaluators: procedures*⟩+≡
```
    function evaluator_sum (eval) result (value)
      complex(default) :: value
```

```
      type(evaluator_t), intent(in) :: eval
      value = interaction_sum (eval%int)
   end function evaluator_sum
```

Renormalize the state matrix by its trace, if nonzero. The renormalization is reflected in the state-matrix norm.

⟨*Evaluators: public*⟩+≡
```
   public :: evaluator_normalize_by_trace
```

⟨*Evaluators: procedures*⟩+≡
```
   subroutine evaluator_normalize_by_trace (eval)
     type(evaluator_t), intent(inout) :: eval
     call interaction_normalize_by_trace (eval%int)
   end subroutine evaluator_normalize_by_trace
```

Analogous, but renormalize by maximal (absolute) value.

⟨*Evaluators: public*⟩+≡
```
   public :: evaluator_normalize_by_max
```

⟨*Evaluators: procedures*⟩+≡
```
   subroutine evaluator_normalize_by_max (eval)
     type(evaluator_t), intent(inout) :: eval
     call interaction_normalize_by_max (eval%int)
   end subroutine evaluator_normalize_by_max
```

Append color-contracted states. Matrix element array and multiplication table are unaffected by this.

⟨*Evaluators: public*⟩+≡
```
   public :: evaluator_add_color_contractions
```

⟨*Evaluators: procedures*⟩+≡
```
   subroutine evaluator_add_color_contractions (eval)
     type(evaluator_t), intent(inout) :: eval
     call interaction_add_color_contractions (eval%int)
   end subroutine evaluator_add_color_contractions
```

Return the quantum-numbers mask of the enclosed interaction.

⟨*Evaluators: public*⟩+≡
```
   public :: evaluator_get_mask
```

⟨*Evaluators: procedures*⟩+≡
```
   function evaluator_get_mask (eval) result (mask)
     type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
     type(evaluator_t), intent(in), target :: eval
     allocate (mask (interaction_get_n_tot (eval%int)))
     mask = interaction_get_mask (eval%int)
   end function evaluator_get_mask
```

Extend the linking of interactions to evaluators.

⟨*Evaluators: public*⟩+≡
```
   public :: interaction_set_source_link
   public :: evaluator_set_source_link
```

⟨*Evaluators: interfaces*⟩+≡

```
    interface interaction_set_source_link
       module procedure interaction_set_source_link_eval
    end interface
    interface evaluator_set_source_link
       module procedure evaluator_set_source_link_int
       module procedure evaluator_set_source_link_eval
    end interface
```

⟨*Evaluators: procedures*⟩+≡

```
    subroutine interaction_set_source_link_eval (int, i, eval1, i1)
      type(interaction_t), intent(inout) :: int
      type(evaluator_t), intent(in), target :: eval1
      integer, intent(in) :: i, i1
      call interaction_set_source_link (int, i, eval1%int, i1)
    end subroutine interaction_set_source_link_eval

    subroutine evaluator_set_source_link_int (eval, i, int1, i1)
      type(evaluator_t), intent(inout) :: eval
      type(interaction_t), intent(in), target :: int1
      integer, intent(in) :: i, i1
      call interaction_set_source_link (eval%int, i, int1, i1)
    end subroutine evaluator_set_source_link_int

    subroutine evaluator_set_source_link_eval (eval, i, eval1, i1)
      type(evaluator_t), intent(inout) :: eval
      type(evaluator_t), intent(in), target :: eval1
      integer, intent(in) :: i, i1
      call interaction_set_source_link (eval%int, i, eval1%int, i1)
    end subroutine evaluator_set_source_link_eval
```

Send momenta to the linked interactions.

⟨*Evaluators: public*⟩+≡

```
    public :: evaluator_receive_momenta
```

⟨*Evaluators: procedures*⟩+≡

```
    subroutine evaluator_receive_momenta (eval)
      type(evaluator_t), intent(inout) :: eval
      call interaction_receive_momenta (eval%int)
    end subroutine evaluator_receive_momenta
```

Reassign external source links from one to another.

⟨*Evaluators: public*⟩+≡

```
    public :: evaluator_reassign_links
```

⟨*Evaluators: interfaces*⟩+≡

```
    interface evaluator_reassign_links
       module procedure evaluator_reassign_links_eval
       module procedure evaluator_reassign_links_int
    end interface
```

⟨*Evaluators: procedures*⟩+≡

```
    subroutine evaluator_reassign_links_eval (eval, eval_src, eval_target)
      type(evaluator_t), intent(inout) :: eval
```

666

```
      type(evaluator_t), intent(in) :: eval_src
      type(evaluator_t), intent(in), target :: eval_target
      if (associated (eval%int_in1)) then
         if (interaction_get_tag (eval%int_in1) &
               == interaction_get_tag (eval_src%int)) then
            eval%int_in1 => eval_target%int
         end if
      end if
      if (associated (eval%int_in2)) then
         if (interaction_get_tag (eval%int_in2) &
               == interaction_get_tag (eval_src%int)) then
            eval%int_in2 => eval_target%int
         end if
      end if
      call interaction_reassign_links (eval%int, eval_src%int, eval_target%int)
   end subroutine evaluator_reassign_links_eval

   subroutine evaluator_reassign_links_int (eval, int_src, int_target)
     type(evaluator_t), intent(inout) :: eval
     type(interaction_t), intent(in) :: int_src
     type(interaction_t), intent(in), target :: int_target
     if (associated (eval%int_in1)) then
        if (interaction_get_tag (eval%int_in1) &
              == interaction_get_tag (int_src)) then
           eval%int_in1 => int_target
        end if
     end if
     if (associated (eval%int_in2)) then
        if (interaction_get_tag (eval%int_in2) &
              == interaction_get_tag (int_src)) then
           eval%int_in2 => int_target
        end if
     end if
     call interaction_reassign_links (eval%int, int_src, int_target)
   end subroutine evaluator_reassign_links_int
```

Return flavor, momentum, and position of the first unstable particle present in the interaction.

⟨*Evaluators: public*⟩+≡
```
   public :: evaluator_get_unstable_particle
```

⟨*Evaluators: procedures*⟩+≡
```
   subroutine evaluator_get_unstable_particle (eval, flv, p, i)
     type(evaluator_t), intent(in) :: eval
     type(flavor_t), intent(out) :: flv
     type(vector4_t), intent(out) :: p
     integer, intent(out) :: i
     call interaction_get_unstable_particle (eval%int, flv, p, i)
   end subroutine evaluator_get_unstable_particle
```

7.7.8   Deleting the evaluator

Only the interaction component needs finalization.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_final
```

⟨*Evaluators: procedures*⟩+≡
```
    elemental subroutine evaluator_final (eval)
      type(evaluator_t), intent(inout) :: eval
      call interaction_final (eval%int)
    end subroutine evaluator_final
```

7.7.9 Evaluation

When the input interactions (which are pointed to in the pairings stored within the evaluator) are filled with values, we can activate the evaluator, i.e., calculate the result values which are stored in the interaction.

The evaluation of matrix elements can be done in parallel. A `forall` construct is not appropriate, however. We would need `do concurrent` here. Nevertheless, the evaluation functions are marked as `pure`.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_evaluate
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_evaluate (eval)
      type(evaluator_t), intent(inout), target :: eval
      integer :: i
      select case (eval%type)
      case (EVAL_PRODUCT)
         do i = 1, size(eval%pairing_array)
            call interaction_evaluate_product (eval%int, i, &
                 eval%int_in1, eval%int_in2, &
                 eval%pairing_array(i)%i1, eval%pairing_array(i)%i2)
         end do
      case (EVAL_SQUARE_WITH_COLOR_FACTORS)
         do i = 1, size(eval%pairing_array)
            call interaction_evaluate_product_cf (eval%int, i, &
                 eval%int_in1, eval%int_in1, &
                 eval%pairing_array(i)%i1, eval%pairing_array(i)%i2, &
                 eval%pairing_array(i)%factor)
         end do
      case (EVAL_SQUARED_FLOWS)
         do i = 1, size(eval%pairing_array)
            call interaction_evaluate_square_c (eval%int, i, &
                 eval%int_in1, &
                 eval%pairing_array(i)%i1)
         end do
      case (EVAL_COLOR_CONTRACTION)
         do i = 1, size(eval%pairing_array)
            call interaction_evaluate_sum (eval%int, i, &
                 eval%int_in1, &
                 eval%pairing_array(i)%i1)
         end do
      end select
    end subroutine evaluator_evaluate
```

7.7.10 Test

Test: Create two interactions. The interactions are twofold connected. The first connection has a helicity index that is kept, the second connection has a helicity index that is summed over. Concatenate the interactions in an evaluator, which thus contains a result interaction. Fill the input interactions with values, activate the evaluator and print the result.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_test
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_test (mdl)
      type(model_t), intent(in), target :: mdl
      call evaluator_test1 (mdl)
    end subroutine evaluator_test

    subroutine evaluator_test1 (mdl)
      type(model_t), intent(in), target :: mdl
      type(interaction_t), target :: int_qqtt, int_tbw
      type(flavor_t), dimension(:), allocatable :: flv
      type(color_t), dimension(:), allocatable :: col
      type(helicity_t), dimension(:), allocatable :: hel
      type(quantum_numbers_t), dimension(:), allocatable :: qn
      integer :: f, c, h1, h2, h3
!     type(vector4_t), dimension(4) :: p
!     type(vector4_t), dimension(2) :: q
      type(quantum_numbers_mask_t) :: qn_mask_conn
      type(evaluator_t), target :: eval
      print *, "*** Evaluator for matrix product"
      print *, "***   Construct interaction for qq -> tt"
      call interaction_init (int_qqtt, 2, 0, 2, set_relations=.true.)
      allocate (flv (4), col (4), hel (4), qn (4))
      do c = 1, 2
         select case (c)
         case (1)
            call color_init_col_acl (col, (/ 1, 0, 1, 0 /), (/ 0, 2, 0, 2 /))
         case (2)
            call color_init_col_acl (col, (/ 1, 0, 2, 0 /), (/ 0, 1, 0, 2 /))
         end select
         do f = 1, 2
            call flavor_init (flv, (/f, -f, 6, -6/), mdl)
            do h1 = -1, 1, 2
               call helicity_init (hel(3), h1)
               do h2 = -1, 1, 2
                  call helicity_init (hel(4), h2)
                  call quantum_numbers_init (qn, flv, col, hel)
                  call interaction_add_state (int_qqtt, qn)
               end do
            end do
         end do
      end do
      call interaction_freeze (int_qqtt)
      deallocate (flv, col, hel, qn)
      print *, "***   Construct interaction for t -> bW"
```

669

```
      call interaction_init (int_tbw, 1, 0, 2, set_relations=.true.)
      allocate (flv (3), col (3), hel (3), qn (3))
      call flavor_init (flv, (/ 6, 5, 24 /), mdl)
      call color_init_col_acl (col, (/ 1, 1, 0 /), (/ 0, 0, 0 /))
      do h1 = -1, 1, 2
         call helicity_init (hel(1), h1)
         do h2 = -1, 1, 2
            call helicity_init (hel(2), h2)
            do h3 = -1, 1
               call helicity_init (hel(3), h3)
               call quantum_numbers_init (qn, flv, col, hel)
               call interaction_add_state (int_tbw, qn)
            end do
         end do
      end do
      call interaction_freeze (int_tbw)
      deallocate (flv, col, hel, qn)
      print *, "***   Link interactions"
      call interaction_set_source_link (int_tbw, 1, int_qqtt, 3)
      qn_mask_conn = new_quantum_numbers_mask (.false.,.false.,.true.)
      print *, "***   Show input"
      call interaction_write (int_qqtt)
      print *
      call interaction_write (int_tbw)
      print *
      print *, "***   Evaluate product"
      call evaluator_init_product &
           (eval, int_qqtt, int_tbw, qn_mask_conn)
      call evaluator_write (eval)

!     p(1) = vector4_moving (1000._default, 1000._default, 3)
!     p(2) = vector4_moving (200._default, 200._default, 2)
!     p(3) = vector4_moving (100._default, 200._default, 1)
!     p(4) = p(1) - p(2) - p(3)
!     call interaction_set_momenta (int1, p)
!     q(1) = vector4_moving (50._default,-50._default, 3)
!     q(2) = p(2) + p(4) - q(1)
!     call interaction_set_momenta (int2, q, outgoing=.true.)
!     call interaction_set_matrix_element &
!          (int1, (/(2._default,0._default), (4._default,1._default), (-3._default,0._default)/))
!     call interaction_set_matrix_element &
!          (int2, (/(-3._default,0._default), (0._default,1._default), (1._default,2._default)/))
!     call evaluator_receive_momenta (eval)
!     call evaluator_evaluate (eval)
!     call interaction_write (int1)
!     print *
!     call interaction_write (int2)
!     print *
!     call evaluator_write (eval)
!     print *
!     call interaction_final (int1)
!     call interaction_final (int2)
!     call evaluator_final (eval)
```

```
!     print *
!     print *, "*** Evaluator for matrix square"
!     call interaction_init (int1, 2, 0, 2, set_relations=.true.)
!     call flavor_init (flv, (/1, -1, 21, 21/), mdl)
!     call color_init (col(1), (/1/))
!     call color_init (col(2), (/-2/))
!     call color_init (col(3), (/2, -3/))
!     call color_init (col(4), (/3, -1/))
!     call quantum_numbers_init (qn, flv, col)
!     call interaction_add_state (int1, qn)
!     call color_init (col(3), (/3, -1/))
!     call color_init (col(4), (/2, -3/))
!     call quantum_numbers_init (qn, flv, col)
!     call interaction_add_state (int1, qn)
!     call color_init (col(3), (/2, -1/))
!     call color_init (col(4), .true.)
!     call quantum_numbers_init (qn, flv, col)
!     call interaction_add_state (int1, qn)
!     call interaction_freeze (int1)
!     ! qn_mask2 = all false (default)
!     call evaluator_init_square (eval, int1, qn_mask2, nc=3)
!     call evaluator_init_squared_flows (eval2, int1, qn_mask2)
!     qn_mask2 = new_quantum_numbers_mask (.false., .true., .true.)
!     call evaluator_init_trace (eval3, eval%int, qn_mask2)
!     call interaction_set_matrix_element &
!          (int1, (/(2._default,0._default), (4._default,1._default), (-3._default,0._default)/))
!     call interaction_set_momenta (int1, p)
!     call interaction_write (int1)
!     print *
!     call evaluator_receive_momenta (eval)
!     call evaluator_evaluate (eval)
!     call evaluator_write (eval)
!     print *
!     call evaluator_receive_momenta (eval2)
!     call evaluator_evaluate (eval2)
!     call evaluator_write (eval2)
!     print *
!     call evaluator_receive_momenta (eval3)
!     call evaluator_evaluate (eval3)
!     call evaluator_write (eval3)
!     call interaction_final (int1)
!     call evaluator_final (eval)
!     call evaluator_final (eval2)
!     call evaluator_final (eval3)
  end subroutine evaluator_test1
```

Chapter 8

Particles

In this chapter, we deal with particles which have well-defined quantum numbers. While within interactions, all correlations are manifest, a particle array is derived by selecting a particular quantum number set. This involves tracing over all other particles, as far as polarization is concerned. Thus, a particle has definite flavor, color, and a single-particle density matrix for polarization.

8.1 Polarization

Particle polarization is determined by a particular quantum state which has just helicity information. For defining polarizations, we adopt the phase convention for a spin-1/2 particle that

$$\rho = \tfrac{1}{2}(1 + \vec{\alpha} \cdot \vec{\sigma}) \tag{8.1}$$

with the polarization axis $\vec{\alpha}$. Using this, we define

1. Trivial polarization: $\vec{\alpha} = 0$. [This is unpolarized, but distinct from the particular undefined polarization matrix which has the same meaning.]

2. Circular polarization: $\vec{\alpha}$ points in $\pm z$ direction.

3. Transversal polarization: $\vec{\alpha}$ points orthogonal to the z direction, with a phase ϕ that is 0 for the x axis, and $\pi/2 = 90°$ for the y axis. For antiparticles, the phase switches sign, corresponding to complex conjugation.

4. Axis polarization, where we explicitly give $\vec{\alpha}$.

For higher spin, we retain this definition, but apply it to the two components with maximum and minimum weight. For massless particles, this is sufficient. For massive particles, we add the possibilities:

5. Longitudinal polarization: Only the 0-component is set. This is possible only for bosons.

6. Diagonal polarization: Explicitly specify all components in the helicity basis.

Obviously, this does not exhaust the possible density matrices for higher spin, but it should cover all practical applications.

⟨polarizations.f90⟩≡

  ⟨*File header*⟩

  module polarizations

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
    use constants, only: imago !NODEP!
  ⟨*Use file utils*⟩
    use lorentz !NODEP!
    use models
    use flavors
    use colors
    use helicities
    use quantum_numbers
    use state_matrices

  ⟨*Standard module head*⟩

  ⟨*Polarizations: public*⟩

  ⟨*Polarizations: types*⟩

  ⟨*Polarizations: interfaces*⟩

  contains

  ⟨*Polarizations: procedures*⟩

  end module polarizations

8.1.1   The polarization type

This is not an extension, but rather a restriction of the quantum state. Flavor and color are ignored, there is just a one-particle helicity density matrix.

⟨*Polarizations: public*⟩≡
    public :: polarization_t

⟨*Polarizations: types*⟩≡
    type :: polarization_t
       logical :: polarized = .false.
       integer :: spin_type = 0
       integer :: multiplicity = 0
       type(state_matrix_t) :: state
    end type polarization_t

8.1.2   Basic initializer and finalizer

We need the particle flavor for determining the allowed helicity values. The density matrix is not set, but prepared to be filled later. This is private.

⟨*Polarizations: procedures*⟩≡
```
    elemental subroutine polarization_init (pol, flv)
      type(polarization_t), intent(out) :: pol
      type(flavor_t), intent(in) :: flv
      pol%spin_type = flavor_get_spin_type (flv)
      pol%multiplicity = flavor_get_multiplicity (flv)
      call state_matrix_init (pol%state, store_values = .true.)
    end subroutine polarization_init
```

The finalizer has to be public. The quantum state contains memory allocated to pointers.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_final
```

⟨*Polarizations: procedures*⟩+≡
```
    elemental subroutine polarization_final (pol)
      type(polarization_t), intent(inout) :: pol
      call state_matrix_final (pol%state)
    end subroutine polarization_final
```

8.1.3 I/O

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_write
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_write (pol, unit)
      type(polarization_t), intent(in) :: pol
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,A,I1,A,I1,A)")  &
           "Polarization: [spin_type = ", pol%spin_type, &
           ", mult = ", pol%multiplicity, "]"
      call state_matrix_write (pol%state, unit=unit)
    end subroutine polarization_write
```

Defined assignment: deep copy

⟨*Polarizations: public*⟩+≡
```
    public :: assignment(=)
```

⟨*Polarizations: interfaces*⟩≡
```
    interface assignment(=)
       module procedure polarization_assign
    end interface
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_assign (pol_out, pol_in)
      type(polarization_t), intent(out) :: pol_out
      type(polarization_t), intent(in) :: pol_in
      pol_out%polarized = pol_in%polarized
      pol_out%spin_type = pol_in%spin_type
```

```
      pol_out%multiplicity = pol_in%multiplicity
      pol_out%state = pol_in%state
    end subroutine polarization_assign
```

Binary I/O.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_write_raw
    public :: polarization_read_raw
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_write_raw (pol, u)
      type(polarization_t), intent(in) :: pol
      integer, intent(in) :: u
      write (u) pol%polarized
      write (u) pol%spin_type
      write (u) pol%multiplicity
      call state_matrix_write_raw (pol%state, u)
    end subroutine polarization_write_raw

    subroutine polarization_read_raw (pol, u, iostat)
      type(polarization_t), intent(out) :: pol
      integer, intent(in) :: u
      integer, intent(out), optional :: iostat
      read (u, iostat=iostat) pol%polarized
      read (u, iostat=iostat) pol%spin_type
      read (u, iostat=iostat) pol%multiplicity
      call state_matrix_read_raw (pol%state, u, iostat=iostat)
    end subroutine polarization_read_raw
```

8.1.4 Accessing contents

Return true if the particle is polarized. This is the case if the first (and only) entry in the quantum state has undefined helicity.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_is_polarized
```

⟨*Polarizations: procedures*⟩+≡
```
    elemental function polarization_is_polarized (pol) result (polarized)
      logical :: polarized
      type(polarization_t), intent(in) :: pol
      polarized = pol%polarized
    end function polarization_is_polarized
```

Return true if the polarization is diagonal, i.e., all entries in the density matrix are diagonal.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_is_diagonal
```

⟨*Polarizations: interfaces*⟩+≡
```
    interface polarization_is_diagonal
       module procedure polarization_is_diagonal0
       module procedure polarization_is_diagonal1
```

```
      end interface
```

⟨*Polarizations: procedures*⟩+≡
```
    function polarization_is_diagonal0 (pol) result (diagonal)
      logical :: diagonal
      type(polarization_t), intent(in) :: pol
      type(state_iterator_t) :: it
      diagonal = .true.
      call state_iterator_init (it, pol%state)
      do while (state_iterator_is_valid (it))
         diagonal = all (quantum_numbers_are_diagonal &
              (state_iterator_get_quantum_numbers (it)))
         if (.not. diagonal) exit
         call state_iterator_advance (it)
      end do
    end function polarization_is_diagonal0

    function polarization_is_diagonal1 (pol) result (diagonal)
      type(polarization_t), dimension(:), intent(in) :: pol
      logical, dimension(size(pol)) :: diagonal
      integer :: i
      do i = 1, size (pol)
         diagonal(i) = polarization_is_diagonal0 (pol(i))
      end do
    end function polarization_is_diagonal1
```

8.1.5   Initialization from state matrix

Here, the state matrix is already known (but not necessarily normalized). The
result will be either unpolarized, or a normalized spin density matrix.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_init_state_matrix
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_init_state_matrix (pol, state)
      type(polarization_t), intent(out) :: pol
      type(state_matrix_t), intent(in), target :: state
      type(state_iterator_t) :: it
      type(flavor_t) :: flv
      type(helicity_t) :: hel
      type(quantum_numbers_t), dimension(1) :: qn
      complex(default) :: value, t
      call state_iterator_init (it, state)
      flv = state_iterator_get_flavor (it, 1)
      hel = state_iterator_get_helicity (it, 1)
      if (helicity_is_defined (hel)) then
         call polarization_init (pol, flv)
         pol%polarized = .true.
         t = 0
         do while (state_iterator_is_valid (it))
            hel = state_iterator_get_helicity (it, 1)
            call quantum_numbers_init (qn(1), hel)
```

```
            value = state_iterator_get_matrix_element (it)
            call state_matrix_add_state (pol%state, qn, value=value)
            if (helicity_is_diagonal (hel))  t = t + value
            call state_iterator_advance (it)
         end do
         call state_matrix_freeze (pol%state)
         if (t /= 0)  call state_matrix_renormalize (pol%state, 1._default / t)
      else
         call polarization_init_unpolarized (pol, flv)
      end if
   end subroutine polarization_init_state_matrix
```

8.1.6 Specific initializers

Unpolarized particle, no helicity labels in the density matrix. The value is
specified as $1/N$, where N is the multiplicity.

Exception: for left-handed or right-handed particles (neutrinos), polarization
is always circular with fraction unity.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_init_unpolarized
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_init_unpolarized (pol, flv)
      type(polarization_t), intent(inout) :: pol
      type(flavor_t), intent(in) :: flv
      type(quantum_numbers_t), dimension(1) :: qn
      complex(default) :: value
      if (flavor_is_left_handed (flv)) then
         call polarization_init_circular (pol, flv, -1._default)
      else if (flavor_is_right_handed (flv)) then
         call polarization_init_circular (pol, flv, 1._default)
      else
         call polarization_init (pol, flv)
         value = 1._default / flavor_get_multiplicity (flv)
         call state_matrix_add_state (pol%state, qn)
         call state_matrix_freeze (pol%state)
         call state_matrix_set_matrix_element (pol%state, value)
      end if
    end subroutine polarization_init_unpolarized
```

Unpolarized particle, but explicit density matrix with helicity states allocated
according to given flavor. Note that fermions have even spin type, bosons odd.
The spin density matrix entries are scaled by `fraction`. This is used for initial-
izing other polarizations:

$$\rho(f) = \frac{|f|}{N}\mathbf{1}.$$

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_init_trivial
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_init_trivial (pol, flv, fraction)
      type(polarization_t), intent(out) :: pol
```

```
    type(flavor_t), intent(in) :: flv
    real(default), intent(in), optional :: fraction
    type(helicity_t) :: hel
    type(quantum_numbers_t), dimension(1) :: qn
    integer :: h, hmax
    logical :: fermion
    complex(default) :: value
    call polarization_init (pol, flv)
    pol%polarized = .true.
    if (present (fraction)) then
       value = fraction / pol%multiplicity
    else
       value = 1._default / pol%multiplicity
    end if
    fermion = mod (pol%spin_type, 2) == 0
    hmax = pol%spin_type / 2
    select case (pol%multiplicity)
    case (1)
       if (flavor_is_left_handed (flv)) then
          call helicity_init (hel, -hmax)
       else if (flavor_is_right_handed (flv)) then
          call helicity_init (hel, hmax)
       else
          call helicity_init (hel, 0)
       end if
       call quantum_numbers_init (qn(1), hel)
       call state_matrix_add_state (pol%state, qn)
    case (2)
       do h = -hmax, hmax, 2*hmax
          call helicity_init (hel, h)
          call quantum_numbers_init (qn(1), hel)
          call state_matrix_add_state (pol%state, qn)
       end do
    case default
       do h = -hmax, hmax
          if (fermion .and. h == 0)  cycle
          call helicity_init (hel, h)
          call quantum_numbers_init (qn(1), hel)
          call state_matrix_add_state (pol%state, qn)
       end do
    end select
    call state_matrix_freeze (pol%state)
    call state_matrix_set_matrix_element (pol%state, value)
  end subroutine polarization_init_trivial
```

The following three modes are useful mainly for spin-1/2 particle and massless
particles of any nonzero spin. Only the highest-weight components are filled.

Circular polarization: The density matrix of the two highest-weight states

is

$$\rho(f) = \frac{1 - |f|}{2} \mathbf{1} + |f| \times \begin{cases} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, & f > 0; \\ \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, & f < 0, \end{cases}$$

If the polarization fraction $|f|$ is unity, we need only one entry in the density matrix.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_init_circular
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_init_circular (pol, flv, fraction)
      type(polarization_t), intent(out) :: pol
      type(flavor_t), intent(in) :: flv
      real(default), intent(in) :: fraction
      type(helicity_t), dimension(2) :: hel
      type(quantum_numbers_t), dimension(1) :: qn
      complex(default) :: value
      integer :: hmax
      call polarization_init (pol, flv)
      pol%polarized = .true.
      hmax = pol%spin_type / 2
      call helicity_init (hel(1), hmax)
      call helicity_init (hel(2),-hmax)
      if (abs (fraction) /= 1) then
         value = (1 + fraction) / 2
         call quantum_numbers_init (qn(1), hel(1))
         call state_matrix_add_state (pol%state, qn, value=value)
         value = (1 - fraction) / 2
         call quantum_numbers_init (qn(1), hel(2))
         call state_matrix_add_state (pol%state, qn, value=value)
      else
         value = abs (fraction)
         if (fraction > 0) then
            call quantum_numbers_init (qn(1), hel(1))
         else
            call quantum_numbers_init (qn(1), hel(2))
         end if
         call state_matrix_add_state (pol%state, qn, value=value)
      end if
      call state_matrix_freeze (pol%state)
    end subroutine polarization_init_circular
```

Transversal polarization is analogous to circular, but we get a density matrix

$$\rho(f, \phi) = \frac{1 - |f|}{2} \mathbf{1} + \frac{|f|}{2} \begin{pmatrix} 1 & e^{-i\phi} \\ e^{i\phi} & 1 \end{pmatrix}.$$

The phase is $\phi = 0$ for the x-axis, $\phi = 90°$ for the y axis as polarization vector. For an antiparticle, the phase switches sign, and for $f < 0$, the off-diagonal elements switch sign.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_init_transversal
```

679

⟨*Polarizations: procedures*⟩+≡
```
subroutine polarization_init_transversal (pol, flv, phi, fraction)
  type(polarization_t), intent(inout) :: pol
  type(flavor_t), intent(in) :: flv
  real(default), intent(in) :: phi, fraction
  call polarization_init_axis &
       (pol, flv, fraction * (/ cos (phi), sin (phi), 0._default/))
end subroutine polarization_init_transversal
```

For axis polarization, we again set only the entries with maximum weight.

$$\rho(f, \phi) = \frac{1}{2} \begin{pmatrix} 1 + \alpha_3 & \alpha_1 - i\alpha_2 \\ \alpha_1 + i\alpha_2 & 1 - \alpha_3 \end{pmatrix}.$$

For an antiparticle, α_2 switches sign (complex conjugate).

⟨*Polarizations: public*⟩+≡
```
public :: polarization_init_axis
```

⟨*Polarizations: procedures*⟩+≡
```
subroutine polarization_init_axis (pol, flv, alpha)
  type(polarization_t), intent(out) :: pol
  type(flavor_t), intent(in) :: flv
  real(default), dimension(3), intent(in) :: alpha
  type(quantum_numbers_t), dimension(1) :: qn
  type(helicity_t), dimension(2,2) :: hel
  complex(default), dimension(2,2) :: value
  integer :: hmax
  call polarization_init (pol, flv)
  pol%polarized = .true.
  hmax = pol%spin_type / 2
  call helicity_init (hel(1,1), hmax, hmax)
  call helicity_init (hel(1,2), hmax,-hmax)
  call helicity_init (hel(2,1),-hmax, hmax)
  call helicity_init (hel(2,2),-hmax,-hmax)
  value(1,1) = (1 + alpha(3)) / 2
  value(2,2) = (1 - alpha(3)) / 2
  if (flavor_is_antiparticle (flv)) then
     value(1,2) = (alpha(1) + imago * alpha(2)) / 2
  else
     value(1,2) = (alpha(1) - imago * alpha(2)) / 2
  end if
  value(2,1) = conjg (value(1,2))
  if (value(1,1) /= 0) then
     call quantum_numbers_init (qn(1), hel(1,1))
     call state_matrix_add_state (pol%state, qn, value=value(1,1))
  end if
  if (value(2,2) /= 0) then
     call quantum_numbers_init (qn(1), hel(2,2))
     call state_matrix_add_state (pol%state, qn, value=value(2,2))
  end if
  if (value(1,2) /= 0) then
     call quantum_numbers_init (qn(1), hel(1,2))
     call state_matrix_add_state (pol%state, qn, value=value(1,2))
     call quantum_numbers_init (qn(1), hel(2,1))
```

```
          call state_matrix_add_state (pol%state, qn, value=value(2,1))
        end if
        call state_matrix_freeze (pol%state)
      end subroutine polarization_init_axis
```

This version specifies the polarization axis in terms of r (polarization degree) and θ, ϕ (polar and azimuthal angles).

If one of the angles is a nonzero multiple of π, roundoff errors typically will result in tiny contributions to unwanted components. Therefore, include a catch for small numbers.

⟨*Polarizations: public*⟩+≡
```
      public :: polarization_init_angles
```

⟨*Polarizations: procedures*⟩+≡
```
      subroutine polarization_init_angles (pol, flv, r, theta, phi)
        type(polarization_t), intent(out) :: pol
        type(flavor_t), intent(in) :: flv
        real(default), intent(in) :: r, theta, phi
        real(default), dimension(3) :: alpha
        real(default), parameter :: eps = 10 * epsilon (1._default)
        alpha(1) = r * sin (theta) * cos (phi)
        alpha(2) = r * sin (theta) * sin (phi)
        alpha(3) = r * cos (theta)
        where (abs (alpha) < eps)  alpha = 0
        call polarization_init_axis (pol, flv, alpha)
      end subroutine polarization_init_angles
```

Longitudinal polarization is defined only for massive bosons. Only the zero component is filled. Otherwise, unpolarized.

⟨*Polarizations: public*⟩+≡
```
      public :: polarization_init_longitudinal
```

⟨*Polarizations: procedures*⟩+≡
```
      subroutine polarization_init_longitudinal (pol, flv, fraction)
        type(polarization_t), intent(out) :: pol
        type(flavor_t), intent(in) :: flv
        real(default), intent(in) :: fraction
        integer :: spin_type, multiplicity
        type(helicity_t) :: hel
        type(quantum_numbers_t), dimension(1) :: qn
        complex(default) :: value
        integer :: n_values
        value = abs (fraction)
        spin_type = flavor_get_spin_type (flv)
        multiplicity = flavor_get_multiplicity (flv)
        if (mod (spin_type, 2) == 1 .and. multiplicity > 2) then
           if (fraction /= 1) then
              call polarization_init_trivial (pol, flv, 1 - fraction)
              n_values = state_matrix_get_n_matrix_elements (pol%state)
              call state_matrix_add_to_matrix_element &
                   (pol%state, n_values/2 + 1, value)
           else
              call polarization_init (pol, flv)
```

```
            pol%polarized = .true.
            call helicity_init (hel, 0)
            call quantum_numbers_init (qn(1), hel)
            call state_matrix_add_state (pol%state, qn)
            call state_matrix_freeze (pol%state)
            call state_matrix_set_matrix_element (pol%state, value)
         end if
      else
         call polarization_init_unpolarized (pol, flv)
      end if
    end subroutine polarization_init_longitudinal
```

This is diagonal polarization: we specify all components explicitly. We use only the positive components. The sum is normalized to unity. We assume that the length of `alpha` is equal to the particle multiplicity.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_init_diagonal
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_init_diagonal (pol, flv, alpha)
      type(polarization_t), intent(inout) :: pol
      type(flavor_t), intent(in) :: flv
      real(default), dimension(:), intent(in) :: alpha
      type(helicity_t) :: hel
      type(quantum_numbers_t), dimension(1) :: qn
      logical, dimension(size(alpha)) :: mask
      real(default) :: norm
      complex(default), dimension(:), allocatable :: value
      logical :: fermion
      integer :: h, hmax, i
      mask = alpha > 0
      norm = sum (alpha, mask);  if (norm == 0)  norm = 1
      allocate (value (count (mask)))
      value = pack (alpha / norm, mask)
      call polarization_init (pol, flv)
      pol%polarized = .true.
      fermion = mod (pol%spin_type, 2) == 0
      hmax = pol%spin_type / 2
      i = 0
      select case (pol%multiplicity)
      case (1)
         if (flavor_is_left_handed (flv)) then
            call helicity_init (hel, -hmax)
         else if (flavor_is_right_handed (flv)) then
            call helicity_init (hel, hmax)
         else
            call helicity_init (hel, 0)
         end if
         call quantum_numbers_init (qn(1), hel)
         call state_matrix_add_state (pol%state, qn)
      case (2)
         do h = -hmax, hmax, 2*hmax
            i = i + 1
            if (mask(i)) then
```

```
                call helicity_init (hel, h)
                call quantum_numbers_init (qn(1), hel)
                call state_matrix_add_state (pol%state, qn)
             end if
          end do
       case default
          do h = -hmax, hmax
             if (fermion .and. h == 0)  cycle
             i = i + 1
             if (mask(i)) then
                call helicity_init (hel, h)
                call quantum_numbers_init (qn(1), hel)
                call state_matrix_add_state (pol%state, qn)
             end if
          end do
       end select
       call state_matrix_freeze (pol%state)
       call state_matrix_set_matrix_element (pol%state, value)
    end subroutine polarization_init_diagonal
```

Generic polarization: we generate all possible density matrix entries, but the values are left zero.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_init_generic
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_init_generic (pol, flv)
      type(polarization_t), intent(out) :: pol
      type(flavor_t), intent(in) :: flv
      type(helicity_t) :: hel
      type(quantum_numbers_t), dimension(1) :: qn
      logical :: fermion
      integer :: hmax, h1, h2
      call polarization_init (pol, flv)
      pol%polarized = .true.
      fermion = mod (pol%spin_type, 2) == 0
      hmax = pol%spin_type / 2
      select case (pol%multiplicity)
      case (1)
         if (flavor_is_left_handed (flv)) then
            call helicity_init (hel, -hmax)
         else if (flavor_is_right_handed (flv)) then
            call helicity_init (hel, hmax)
         else
            call helicity_init (hel, 0)
         end if
         call quantum_numbers_init (qn(1), hel)
         call state_matrix_add_state (pol%state, qn)
      case (2)
         do h1 = -hmax, hmax, 2*hmax
            do h2 = -hmax, hmax, 2*hmax
               call helicity_init (hel, h1, h2)
               call quantum_numbers_init (qn(1), hel)
               call state_matrix_add_state (pol%state, qn)
```

```
            end do
         end do
      case default
         do h1 = -hmax, hmax
            if (fermion .and. h1 == 0)  cycle
            do h2 = -hmax, hmax
               if (fermion .and. h2 == 0)  cycle
               call helicity_init (hel, h1, h2)
               call quantum_numbers_init (qn(1), hel)
               call state_matrix_add_state (pol%state, qn)
            end do
         end do
      end select
      call state_matrix_freeze (pol%state)
   end subroutine polarization_init_generic
```

8.1.7 Operations

Combine polarization states by computing the outer product of the state matrices.

⟨*Polarizations: public*⟩+≡
```
   public :: combine_polarization_states
```

⟨*Polarizations: procedures*⟩+≡
```
   subroutine combine_polarization_states (pol, state)
     type(polarization_t), dimension(:), intent(in), target :: pol
     type(state_matrix_t), intent(out) :: state
     call outer_multiply (pol%state, state)
   end subroutine combine_polarization_states
```

Transform a polarization density matrix into a polarization vector. This is possible without information loss only for spin-1/2 and for massless particles. To get a unique answer in all cases, we consider only the components with highest weight. Obviously, this loses the longitudinal component of a massive vector, for instance.

This is the inverse operation of `polarization_init_axis` above, where the polarization fraction is set to unity.

⟨*Polarizations: public*⟩+≡
```
   public :: polarization_get_axis
```

⟨*Polarizations: procedures*⟩+≡
```
   function polarization_get_axis (pol) result (alpha)
     real(default), dimension(3) :: alpha
     type(polarization_t), intent(in) :: pol
     type(state_iterator_t) :: it
     complex(default), dimension(2,2) :: value
     type(helicity_t), dimension(2,2) :: hel
     type(helicity_t), dimension(1) :: hel1
     integer :: hmax, i, j
     if (pol%polarized) then
        hmax = pol%spin_type / 2
        call helicity_init (hel(1,1), hmax, hmax)
```

```
       call helicity_init (hel(1,2), hmax,-hmax)
       call helicity_init (hel(2,1),-hmax, hmax)
       call helicity_init (hel(2,2),-hmax,-hmax)
       value = 0
       call state_iterator_init (it, pol%state)
       do while (state_iterator_is_valid (it))
          hel1 = state_iterator_get_helicity (it)
          SCAN_HEL: do i = 1, 2
             do j = 1, 2
                if (hel1(1) == hel(i,j)) then
                   value(i,j) = state_iterator_get_matrix_element (it)
                   exit SCAN_HEL
                end if
             end do
          end do SCAN_HEL
          call state_iterator_advance (it)
       end do
       alpha(1) = value(1,2) + value(2,1)
       alpha(2) = imago * (value(1,2) - value(2,1))
       alpha(3) = value(1,1) - value(2,2)
    else
       alpha = 0
    end if
  end function polarization_get_axis
```

This function returns polarization degree and polar and azimuthal angles (θ, ϕ) of the polarization axis.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_to_angles
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_to_angles (pol, r, theta, phi)
      type(polarization_t), intent(in) :: pol
      real(default), intent(out) :: r, theta, phi
      real(default), dimension(3) :: alpha
      real(default) :: r12
      if (pol%polarized) then
         alpha = polarization_get_axis (pol)
         r = sqrt (sum (alpha**2))
         if (any (alpha /= 0)) then
            r12 = sqrt (alpha(1)**2 + alpha(2)**2)
            theta = atan2 (r12, alpha(3))
            if (any (alpha(1:2) /= 0)) then
               phi = atan2 (alpha(2), alpha(1))
            else
               phi = 0
            end if
         else
            theta = 0
         end if
      else
         r = 0
         theta = 0
         phi = 0
```

```
      end if
    end subroutine polarization_to_angles
```

8.1.8   Test

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_test
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_test
      use os_interface, only: os_data_t
      type(os_data_t) :: os_data
      type(model_t), pointer :: model
      type(polarization_t) :: pol
      type(flavor_t) :: flv
      real(default), dimension(3) :: alpha
      real(default) :: r, theta, phi
      print *, "* Read model file"
      call syntax_model_file_init ()
      call model_list_read_model &
          (var_str("QCD"), var_str("test.mdl"), os_data, model)
      print *, "Unpolarized fermion"
      call flavor_init (flv, 1, model)
      call polarization_init_unpolarized (pol, flv)
      call polarization_write (pol)
      print *, "diagonal =", polarization_is_diagonal (pol)
      call polarization_final (pol)
      print *, "Unpolarized fermion"
      call polarization_init_circular (pol, flv, 0._default)
      call polarization_write (pol)
      call polarization_final (pol)
      print *, "Transversally polarized fermion, phi=0"
      call polarization_init_transversal (pol, flv, 0._default, 1._default)
      call polarization_write (pol)
      print *, "diagonal =", polarization_is_diagonal (pol)
      call polarization_final (pol)
      print *, "Transversally polarized fermion, phi=0.9, frac=0.8"
      call polarization_init_transversal (pol, flv, 0.9_default, 0.8_default)
      call polarization_write (pol)
      print *, "diagonal =", polarization_is_diagonal (pol)
      call polarization_final (pol)
      print *, "All polarization directions of a fermion"
      call polarization_init_generic (pol, flv)
      call polarization_write (pol)
      call polarization_final (pol)
      call flavor_init (flv, 21, model)
      print *, "Circularly polarized gluon, frac=0.3"
      call polarization_init_circular (pol, flv, 0.3_default)
      call polarization_write (pol)
      call polarization_final (pol)
      call flavor_init (flv, 23, model)
      print *, "Circularly polarized massive vector, frac=-0.7"
      call polarization_init_circular (pol, flv,  -0.7_default)
```

```
      call polarization_write (pol)
      call polarization_final (pol)
      print *, "Circularly polarized massive vector"
      call polarization_init_circular (pol, flv, 1._default)
      call polarization_write (pol)
      call polarization_final (pol)
      print *, "Longitudinally polarized massive vector, frac=0.4"
      call polarization_init_longitudinal (pol, flv, 0.4_default)
      call polarization_write (pol)
      call polarization_final (pol)
      print *, "Longitudinally polarized massive vector"
      call polarization_init_longitudinal (pol, flv, 1._default)
      call polarization_write (pol)
      call polarization_final (pol)
      print *, "Diagonally polarized massive vector"
      call polarization_init_diagonal &
           (pol, flv, (/0._default, 1._default, 2._default/))
      call polarization_write (pol)
      call polarization_final (pol)
      print *, "All polarization directions of a massive vector"
      call polarization_init_generic (pol, flv)
      call polarization_write (pol)
      call polarization_final (pol)
      call flavor_init (flv, 21, model)
      print *, "Axis polarization (0.2, 0.4, 0.6)"
      alpha = (/0.2_default, 0.4_default, 0.6_default/)
      call polarization_init_axis (pol, flv, alpha)
      call polarization_write (pol)
      print *, "Recovered axis:"
      alpha = polarization_get_axis (pol)
      print *, "Angle polarization (0.5, 0.6, -1)"
      r = 0.5_default
      theta = 0.6_default
      phi = -1._default
      call polarization_init_angles (pol, flv, r, theta, phi)
      call polarization_write (pol)
      print *, "Recovered parameters (r, theta, phi):"
      call polarization_to_angles (pol, r, theta, phi)
      print *, r, theta, phi
      call polarization_final (pol)
    end subroutine polarization_test
```

8.2 Event formats

This section provides the interface to event formats for user-defined processes,
specifically the Les Houches Accord format and the HEPEVT format.

⟨event_formats.f90⟩≡
  ⟨*File header*⟩

```
  module event_formats
```

⟨*Use kinds*⟩
```
  use kinds, only: i32, i64 !NODEP!
  use constants, only: pb_per_fb !NODEP!
```
⟨*Use file utils*⟩
```
  use lorentz !NODEP!
  use prt_lists
  use flavors
  use colors
  use helicities
  use quantum_numbers
  use polarizations
  use stdhep_interface
```

⟨*Standard module head*⟩

⟨*Event formats: public*⟩

⟨*Event formats: parameters*⟩

⟨*Event formats: variables*⟩

⟨*Event formats: common blocks*⟩

⟨*Event formats: interfaces*⟩

```
contains
```

⟨*Event formats: procedures*⟩

```
end module event_formats
```

8.2.1   Les Houches Event File: header/footer

These two routines write the header and footer for the Les Houches Event File format (LHEF).

The current version writes no information except for the generator name and version.

⟨*Event formats: public*⟩≡
```
    public :: les_houches_events_write_header
    public :: les_houches_events_write_footer
```
⟨*Event formats: procedures*⟩≡
```
    subroutine les_houches_events_write_header (unit)
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, *) '<LesHouchesEvents version="1.0">'
      write (u, *) '<header>'
      write (u, *) '  <generator_name>WHIZARD</generator_name>'
      write (u, *) '  <generator_version>⟨Version⟩</generator_version>'
      write (u, *) '</header>'
    end subroutine les_houches_events_write_header

    subroutine les_houches_events_write_footer (unit)
```

688

```
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit);  if (u < 0)  return
    write (u, *) '</LesHouchesEvents>'
  end subroutine les_houches_events_write_footer
```

8.2.2 The HEPRUP common block

This common block is filled once per run.

Run characteristics

The maximal number of different processes.

⟨*Event formats: parameters*⟩≡
```
    integer, parameter :: MAXPUP = 100
```

The beam PDG codes.

⟨*Event formats: variables*⟩≡
```
    integer, dimension(2) :: IDBMUP
```

The beam energies in GeV.

⟨*Event formats: variables*⟩+≡
```
    double precision, dimension(2) :: EBMUP
```

The PDF group and set for the two beams. (Undefined: use -1; LHAPDF: use group $= 0$).

⟨*Event formats: variables*⟩+≡
```
    integer, dimension(2) :: PDFGUP
    integer, dimension(2) :: PDFSUP
```

The (re)weighting model. 1: events are weighted, the shower generator (SHG) selects processes according to the maximum weight (in pb) and unweights events. 2: events are weighted, the SHG selects processes according to their cross section (in pb) and unweights events. 3: events are unweighted and simply run through the SHG. 4: events are weighted, and the SHG keeps the weight. Negative numbers: negative weights are allowed (and are reweighted to ± 1 by the SHG, if allowed).

WHIZARD only supports modes 3 and 4, as the SHG is not given control over process selection. This is consistent with writing events to file, for offline showering.

⟨*Event formats: variables*⟩+≡
```
    integer :: IDWTUP
```

The number of different processes.

⟨*Event formats: variables*⟩+≡
```
    integer :: NPRUP
```

Process characteristics

Cross section and error in pb. (Cross section is needed only for IDWTUP $= 2$, so here both values are given for informational purposes only.)

⟨*Event formats: variables*⟩+≡
```
    double precision, dimension(MAXPUP) :: XSECUP
    double precision, dimension(MAXPUP) :: XERRUP
```

Maximum weight, i.e., the maximum value that `XWGTUP` can take. Also unused for the supported weighting models. It is ±1 for unweighted events.

⟨*Event formats: variables*⟩+≡
```
double precision, dimension(MAXPUP) :: XMAXUP
```

Internal ID of the selected process, matches `IDPRUP` below.

⟨*Event formats: variables*⟩+≡
```
integer, dimension(MAXPUP) :: LPRUP
```

The common block

⟨*Event formats: common blocks*⟩≡
```
common /HEPRUP/ &
     IDBMUP, EBMUP, PDFGUP, PDFSUP, IDWTUP, NPRUP, &
     XSECUP, XERRUP, XMAXUP, LPRUP
save /HEPRUP/
```

Fill the run characteristics of the common block. The initialization sets the beam properties, number of processes, and weighting model.

⟨*Event formats: public*⟩+≡
```
public :: heprup_init
```

⟨*Event formats: procedures*⟩+≡
```
subroutine heprup_init &
     (beam_pdg, beam_energy, n_processes, unweighted, negative_weights)
  integer, dimension(2), intent(in) :: beam_pdg
  real(default), dimension(2), intent(in) :: beam_energy
  integer, intent(in) :: n_processes
  logical, intent(in) :: unweighted
  logical, intent(in) :: negative_weights
  IDBMUP = beam_pdg
  EBMUP = beam_energy
  PDFGUP = -1
  PDFSUP = -1
  if (unweighted) then
     IDWTUP = 3
  else
     IDWTUP = 4
  end if
  if (negative_weights)  IDWTUP = - IDWTUP
  NPRUP = n_processes
end subroutine heprup_init
```

Specify PDF set info. Since we support only LHAPDF, the group entry is zero.

⟨*Event formats: public*⟩+≡
```
public :: heprup_set_lhapdf_id
```

⟨*Event formats: procedures*⟩+≡
```
subroutine heprup_set_lhapdf_id (i_beam, pdf_id)
  integer, intent(in) :: i_beam, pdf_id
  PDFGUP(i_beam) = 0
  PDFSUP(i_beam) = pdf_id
end subroutine heprup_set_lhapdf_id
```

Fill the characteristics for a particular process. Only the process ID is mandatory. Note that WHIZARD computes cross sections in fb, so we have to rescale to pb. The maximum weight is meaningless for unweighted events.

⟨*Event formats: public*⟩+≡
```
public :: heprup_set_process_parameters
```

⟨*Event formats: procedures*⟩+≡
```
subroutine heprup_set_process_parameters &
     (i, process_id, cross_section, error, max_weight)
  integer, intent(in) :: i, process_id
  real(default), intent(in), optional :: cross_section, error, max_weight
  LPRUP(i) = process_id
  if (present (cross_section)) then
     XSECUP(i) = cross_section * pb_per_fb
  else
     XSECUP(i) = 0
  end if
  if (present (error)) then
     XERRUP(i) = error * pb_per_fb
  else
     XERRUP(i) = 0
  end if
  select case (IDWTUP)
  case (3);  XMAXUP(i) = 1
  case (4)
     if (present (max_weight)) then
        XMAXUP(i) = max_weight * pb_per_fb
     else
        XMAXUP(i) = 0
     end if
  end select
end subroutine heprup_set_process_parameters
```

8.2.3 Run parameter output

This routine writes the initialization block according to the LHEF standard. It uses the current contents of the HEPRUP block.

⟨*Event formats: public*⟩+≡
```
public :: heprup_write_lhef
```

⟨*Event formats: procedures*⟩+≡
```
subroutine heprup_write_lhef (unit)
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit);  if (u < 0)  return
  write (u, *) "<init>"
  write (u, *) IDBMUP, EBMUP, PDFGUP, PDFSUP, IDWTUP, NPRUP
  do i = 1, NPRUP
     write (u, *) XSECUP(i), XERRUP(i), XMAXUP(i), LPRUP(i)
  end do
  write (u, *) "</init>"
end subroutine heprup_write_lhef
```

8.2.4 The HEPEUP And HEPEVT common block

These common blocks are filled once per event.

Event characteristics

The maximal number of particles in an event record.

⟨*Event formats: parameters*⟩+≡
```
integer, parameter :: MAXNUP = 500
```

The number of particles in this event.

⟨*Event formats: variables*⟩+≡
```
integer :: NUP
```

The process ID for this event.

⟨*Event formats: variables*⟩+≡
```
integer :: IDPRUP
```

The weight of this event (± 1 for unweighted events).

⟨*Event formats: variables*⟩+≡
```
double precision :: XWGTUP
```

The factorization scale that is used for PDF calculation (-1 if undefined).

⟨*Event formats: variables*⟩+≡
```
double precision :: SCALUP
```

The QED and QCD couplings α used for this event (-1 if undefined).

⟨*Event formats: variables*⟩+≡
```
double precision :: AQEDUP
double precision :: AQCDUP
```

Particle characteristics

The PDG code:

⟨*Event formats: variables*⟩+≡
```
integer, dimension(MAXNUP) :: IDUP
```

The status code. Incoming: -1, outgoing: $+1$. Intermediate t-channel propagator: -2 (currently not used by WHIZARD). Intermediate resonance whose mass should be preserved: 2. Intermediate resonance for documentation: 3 (currently not used). Beam particles: -9.

⟨*Event formats: variables*⟩+≡
```
integer, dimension(MAXNUP) :: ISTUP
```

Index of first and last mother.

⟨*Event formats: variables*⟩+≡
```
integer, dimension(2,MAXNUP) :: MOTHUP
```

Color line index of the color and anticolor entry for the particle. The standard recommends using large numbers; we start from MAXNUP+1.

⟨*Event formats: variables*⟩+≡
```
integer, dimension(2,MAXNUP) :: ICOLUP
```

Momentum, energy, and invariant mass: (p_x, p_y, p_z, E, M). For space-like particles, M is the negative square root of the absolute value of the invariant mass.

⟨*Event formats: variables*⟩+≡
```
double precision, dimension(5,MAXNUP) :: PUP
```

Invariant lifetime (distance) from production to decay in mm.

⟨Event formats: variables⟩+≡
```
    double precision, dimension(MAXNUP) :: VTIMUP
```

Cosine of the angle between the spin-vector and a particle and the 3-momentum of its mother, given in the lab frame. If undefined/unpolarized: 9.

⟨Event formats: variables⟩+≡
```
    double precision, dimension(MAXNUP) :: SPINUP
```

The common blocks

⟨Event formats: common blocks⟩+≡
```
    common /HEPEUP/ &
        NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP, &
        IDUP, ISTUP, MOTHUP, ICOLUP, PUP, VTIMUP, SPINUP
    save /HEPEUP/
```

Fill the event characteristics of the common block. The initialization sets only the number of particles and initializes the rest with default values. The other routine sets the optional parameters.

⟨Event formats: public⟩+≡
```
    public :: hepeup_init
    public :: hepeup_set_event_parameters
```

⟨Event formats: procedures⟩+≡
```
    subroutine hepeup_init (n_tot)
      integer, intent(in) :: n_tot
      NUP = n_tot
      IDPRUP = 0
      XWGTUP = 1
      SCALUP = -1
      AQEDUP = -1
      AQCDUP = -1
    end subroutine hepeup_init

    subroutine hepeup_set_event_parameters &
        (proc_id, weight, scale, alpha_qed, alpha_qcd)
      integer, intent(in), optional :: proc_id
      real(default), intent(in), optional :: weight, scale, alpha_qed, alpha_qcd
      if (present (proc_id))   IDPRUP = proc_id
      if (present (weight))    XWGTUP = weight
      if (present (scale))     SCALUP = scale
      if (present (alpha_qed)) AQEDUP = alpha_qed
      if (present (alpha_qcd)) AQCDUP = alpha_qcd
    end subroutine hepeup_set_event_parameters
```

Below we need the particle status codes which are actually defined in the `prt_lists` module.

Set the entry for a specific particle. All parameters are set with the exception of lifetime and spin, where default values are stored.

⟨Event formats: public⟩+≡
```
    public :: hepeup_set_particle
```

693

⟨*Event formats: procedures*⟩+≡
```
    subroutine hepeup_set_particle (i, pdg, status, parent, col, p, m2)
      integer, intent(in) :: i
      integer, intent(in) :: pdg, status
      integer, dimension(:), intent(in) :: parent
      type(vector4_t), intent(in) :: p
      integer, dimension(2), intent(in) :: col
      real(default), intent(in) :: m2
      IDUP(i) = pdg
      select case (status)
      case (PRT_BEAM);      ISTUP(i) = -9
      case (PRT_INCOMING);  ISTUP(i) = -1
      case (PRT_OUTGOING);  ISTUP(i) =  1
      case (PRT_RESONANT);  ISTUP(i) =  2
      case default;         ISTUP(i) =  0
      end select
      select case (size (parent))
      case (1);    MOTHUP(:,i) = parent(1)
      case (2);    MOTHUP(:,i) = parent
      case default; MOTHUP(:,i) = 0
      end select
      if (col(1) > 0) then
         ICOLUP(1,i) = 500 + col(1)
      else
         ICOLUP(1,i) = 0
      end if
      if (col(2) > 0) then
         ICOLUP(2,i) = 500 + col(2)
      else
         ICOLUP(2,i) = 0
      end if
      PUP(1:3,i) = vector3_get_components (space_part (p))
      PUP(4,i) = energy (p)
      PUP(5,i) = sign (sqrt (abs (m2)), m2)
      VTIMUP(i) = 0
      SPINUP(i) = 9
    end subroutine hepeup_set_particle
```

Set the lifetime, actually $c\tau$ measured im mm, where τ is the invariant lifetime.

⟨*Event formats: public*⟩+≡
```
    public :: hepeup_set_particle_lifetime
```

⟨*Event formats: procedures*⟩+≡
```
    subroutine hepeup_set_particle_lifetime (i, lifetime)
      integer, intent(in) :: i
      real(default), intent(in) :: lifetime
      VTIMUP(i) = lifetime
    end subroutine hepeup_set_particle_lifetime
```

Set the particle spin entry. We need the cosine of the angle of the spin axis with respect to the three-momentum of the parent particle.

If the particle has a full polarization density matrix given, we need the particle momentum and polarization as well as the mother-particle momentum.

The polarization is transformed into a spin vector (which is sensible only for spin-1/2 or massless particles), which then is transformed into the lab frame (by a rotation of the 3-axis to the particle momentum axis). Finally, we compute the scalar product of this vector with the mother-particle three-momentum.

This puts severe restrictions on the applicability of this definition, and Lorentz invariance is lost. Unfortunately, the Les Houches Accord requires this computation.

⟨*Event formats: public*⟩+≡
```
    public :: hepeup_set_particle_spin
```

⟨*Event formats: interfaces*⟩≡
```
    interface hepeup_set_particle_spin
       module procedure hepeup_set_particle_spin_pol
    end interface
```

⟨*Event formats: procedures*⟩+≡
```
    subroutine hepeup_set_particle_spin_pol (i, p, pol, p_mother)
      integer, intent(in) :: i
      type(vector4_t), intent(in) :: p
      type(polarization_t), intent(in) :: pol
      type(vector4_t), intent(in) :: p_mother
      type(vector3_t) :: s3, p3
      type(vector4_t) :: s4
      s3 = vector3_moving (polarization_get_axis (pol))
      p3 = space_part (p)
      s4 = rotation_to_2nd (3, p3) * vector4_moving (0._default, s3)
      SPINUP(i) = enclosed_angle_ct (s4, p_mother)
    end subroutine hepeup_set_particle_spin_pol
```

The HEPEVT common block

For the LEP Monte Carlos, a standard common block has been proposed in AKV89. We strongly recommend its use. (The description is an abbreviated transcription of AKV89, Vol. 3, pp. 327-330).

NMXHEP is the maximum number of entries:

⟨*Event formats: variables*⟩+≡
```
    integer, parameter :: NMXHEP = 4000
```

NEVHEP is normally the event number, but may take special values as follows:
0 the program does not keep track of event numbers. -1 a special initialization record. -2 a special final record.

⟨*Event formats: variables*⟩+≡
```
    integer :: NEVHEP
```

NHEP holds the number of entries for this event.

⟨*Event formats: variables*⟩+≡
```
    integer :: NHEP
```

The entry `ISTHEP(N)` gives the status code for the `Nth` entry, with the following semantics: 0 a null entry. 1 an existing entry, which has not decayed or fragmented. 2 a decayed or fragmented entry, which is retained for event history information. 3 documentation line. 4- 10 reserved for future standards. 11-200 at the disposal of each model builder. 201- at the disposal of users.

⟨*Event formats: variables*⟩+≡
```
integer, dimension(NMXHEP) :: ISTHEP
```

The Particle Data Group has proposed standard particle codes, which are to be stored in `IDHEP(N)`.

⟨*Event formats: variables*⟩+≡
```
integer, dimension(NMXHEP) :: IDHEP
```

`JMOHEP(1,N)` points to the mother of the `Nth` entry, if any. It is set to zero for initial entries. `JMOHEP(2,N)` points to the second mother, if any.

⟨*Event formats: variables*⟩+≡
```
integer, dimension(2, NMXHEP) :: JMOHEP
```

`JDAHEP(1,N)` and `JDAHEP(2,N)` point to the first and last daughter of the `Nth` entry, if any. These are zero for entries which have not yet decayed. The other daughters are stored in between these two.

⟨*Event formats: variables*⟩+≡
```
integer, dimension(2, NMXHEP) :: JDAHEP
```

In `PHEP` we store the momentum of the particle, more specifically this means that `PHEP(1,N)`, `PHEP(2,N)`, and `PHEP(3,N)` contain the momentum in the x, y, and z direction (as defined by the machine people), measured in GeV/c. `PHEP(4,N)` contains the energy in GeV and `PHEP(5,N)` the mass in GeV/c^2. The latter may be negative for spacelike partons.

⟨*Event formats: variables*⟩+≡
```
double precision, dimension(5, NMXHEP) :: PHEP
```

Finally `VHEP` is the place to store the position of the production vertex. `VHEP(1,N)`, `VHEP(2,N)`, and `VHEP(3,N)` contain the x, y, and z coordinate (as defined by the machine people), measured in mm. `VHEP(4,N)` contains the production time in mm/c.

⟨*Event formats: variables*⟩+≡
```
double precision, dimension(4, NMXHEP) :: VHEP
```

As an amendment to the proposed standard common block HEPEVT, we also have a polarisation common block HEPSPN, as described in AKV89. `SHEP(1,N)`, `SHEP(2,N)`, and `SHEP(3,N)` give the x, y, and z component of the spinvector s of a fermion in the fermions restframe.

Furthermore, we add the polarization of the corresponding outgoing particles:

⟨*Event formats: variables*⟩+≡
```
integer, dimension(NMXHEP) :: hepevt_pol
```

By convention, `SHEP(4,N)` is always 1. All this is taken from StdHep 4.06 manual and written using Fortran90 conventions.

⟨*Event formats: common blocks*⟩+≡
```
common /HEPEVT/ &
      NEVHEP, NHEP, ISTHEP, IDHEP, &
      JMOHEP, JDAHEP, PHEP, VHEP
save /HEPEVT/
```

Here we store HEPEVT parameters of the WHIZARD 1 realization which are not part of the HEPEVT common block.

⟨*Event formats: variables*⟩+≡
```
integer :: hepevt_n_out, hepevt_n_remnants
```

⟨*Event formats: variables*⟩+≡
```
double precision :: hepevt_weight, hepevt_function_value
double precision :: hepevt_function_ratio
```

Filling HEPEVT: If the event count is not provided, set `NEVHEP` to zero. If the event count is `-1` or `-2`, the record corresponds to initialization and finalization, and the event is irrelevant.

Note that the event count may be larger than 2^{31} (2 GEvents). In that case, cut off the upper bits since `NEVHEP` is probably limited to default integer.

⟨*Event formats: public*⟩+≡
```
public :: hepevt_init
public :: hepevt_set_event_parameters
```

⟨*Event formats: procedures*⟩+≡
```
subroutine hepevt_init (n_tot, n_out)
  integer, intent(in) :: n_tot, n_out
  NHEP            = n_tot
  NEVHEP         = 0
  hepevt_n_out      = n_out
  hepevt_n_remnants = 0
  hepevt_weight     = 1
  hepevt_function_value = 0
  hepevt_function_ratio = 1
end subroutine hepevt_init

subroutine hepevt_set_event_parameters &
      (n_tot, n_out, n_remnants, weight, function_value, &
       function_ratio, i_evt)
  integer, intent(in), optional :: n_tot, n_out, n_remnants, i_evt
  real(default), intent(in), optional :: weight, function_value, &
      function_ratio
  integer(i32), parameter :: huge32 = huge (0_i32)
  if (present (n_tot)) NHEP = n_tot
  if (present (i_evt)) NEVHEP = i_evt
  if (present (n_out)) hepevt_n_out = n_out
  if (present (n_remnants)) hepevt_n_remnants = n_remnants
  if (present (weight)) hepevt_weight = weight
  if (present (function_value)) hepevt_function_value = &
       function_value
```

```
      if (present (function_ratio)) hepevt_function_ratio = &
           function_ratio
    end subroutine hepevt_set_event_parameters
```

Set the entry for a specific particle. All parameters are set with the exception of lifetime and spin, where default values are stored.

⟨*Event formats: public*⟩+≡
```
    public :: hepevt_set_particle
```

⟨*Event formats: procedures*⟩+≡
```
    subroutine hepevt_set_particle (i, pdg, status, parent,  &
           children, p, m2, hel)
      integer, intent(in) :: i
      integer, intent(in) :: pdg, status
      integer, dimension(:), intent(in) :: parent
      integer, dimension(:), intent(in) :: children
      type(vector4_t), intent(in) :: p
      real(default), intent(in) :: m2
      integer, intent(in) :: hel
      IDHEP(i) = pdg
      select case (status)
        case (PRT_BEAM);     ISTHEP(i) = 2
        case (PRT_INCOMING); ISTHEP(i) = 2
        case (PRT_OUTGOING); ISTHEP(i) = 1
        case (PRT_RESONANT); ISTHEP(i) = 2
        case default;        ISTHEP(i) = 0
      end select
      select case (size (parent))
      case (1);    JMOHEP(:,i) = parent(1)
      case (2);    JMOHEP(:,i) = parent
      case default; JMOHEP(:,i) = 0
      end select
      select case (status)
        case (PRT_OUTGOING); JDAHEP(:,i) = 0
        case (PRT_BEAM,PRT_INCOMING,PRT_RESONANT)
           JDAHEP(1,i) = children(1);
           JDAHEP(2,i) = children(size (children));
        case default;    JDAHEP(:,i) = 0
      end select
      PHEP(1:3,i) = vector3_get_components (space_part (p))
      PHEP(4,i) = energy (p)
      PHEP(5,i) = sign (sqrt (abs (m2)), m2)
      VHEP(1:4,i) = 0
      hepevt_pol(i) = hel
    end subroutine hepevt_set_particle
```

8.2.5 Event output

This routine writes event output according to the LHEF standard. It uses the current contents of the HEPEUP block.

⟨*Event formats: public*⟩+≡
```
    public :: hepeup_write_lhef
```

```
    public :: hepeup_write_lha
```
⟨*Event formats: procedures*⟩+≡
```
    subroutine hepeup_write_lhef (unit)
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, *) "<event>"
      write (u, *) NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP
      do i = 1, NUP
         write (u, *) IDUP(i), ISTUP(i), MOTHUP(:,i), ICOLUP(:,i), &
              PUP(:,i), VTIMUP(i), SPINUP(i)
      end do
      write (u, *) "</event>"
    end subroutine hepeup_write_lhef

    subroutine hepeup_write_lha (unit)
      integer, intent(in), optional :: unit
      integer :: u, i
      integer, dimension(MAXNUP) :: spin_up
      spin_up = SPINUP
      u = output_unit (unit);  if (u < 0)  return
 16 format(2(1x,I5),1x,F17.10,3(1x,F13.6))
 17 format(500(1x,I5))
 18 format(1x,I5,4(1x,F17.10))
      write (u, 16) NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP
      write (u, 17) IDUP(:NUP)
      write (u, 17) MOTHUP(1,:NUP)
      write (u, 17) MOTHUP(2,:NUP)
      write (u, 17) ICOLUP(1,:NUP)
      write (u, 17) ICOLUP(2,:NUP)
      write (u, 17) ISTUP(:NUP)
      write (u, 17) spin_up(:NUP)
      do i = 1, NUP
            write (u, 18) i, PUP((/ 4,1,2,3 /), i)
      end do

    end subroutine hepeup_write_lha
```

This routine writes event output according to the HEPEVT standard. It uses
the current contents of the HEPEVT block and some additional parameters
according to the standard in WHIZARD 1. For the long ASCII format, the value
of the sample function (i.e. the product of squared matrix element, structure
functions and phase space factor is printed out). The option of reweighting
matrix elements with respect to some reference cross section is not implemented
in WHIZARD 2, therefore the second entry in the long ASCII format (the
function ratio) is always one. The ATHENA format is an implementation of the
HEPEVT format that is readable by the ATLAS ATHENA software framework.
It is very similar to the WHIZARD 1 HEPEVT format, except that it contains
an event counter, a particle counter inside the event, and has the HEPEVT
ISTHEP status before the PDG code.

⟨*Event formats: public*⟩+≡
```
    public :: hepevt_write_hepevt
```

```
      public :: hepevt_write_ascii
      public :: hepevt_write_athena
```
⟨*Event formats: procedures*⟩+≡
```
    subroutine hepevt_write_hepevt (unit)
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, *) NHEP, hepevt_n_out, hepevt_n_remnants, hepevt_weight
      do i = 1, NHEP
         write (u, *) ISTHEP(i), IDHEP(i), JMOHEP(:,i), JDAHEP(:,i), &
            hepevt_pol(i)
         write (u, *) PHEP(:,i)
         write (u, *) VHEP(:,i), 0.d0
      end do
    end subroutine hepevt_write_hepevt

    subroutine hepevt_write_ascii (unit, long)
      integer, intent(in), optional :: unit
      logical, intent(in) :: long
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, *) NHEP, hepevt_n_out, hepevt_n_remnants, hepevt_weight
      do i = 1, NHEP
         write (u, *) IDHEP(i), hepevt_pol(i)
         write (u, *) PHEP(:,i)
      end do
      if (long) write (u, *) hepevt_function_value, hepevt_function_ratio
    end subroutine hepevt_write_ascii

    subroutine hepevt_write_athena (unit, i_evt)
      integer, intent(in), optional :: unit, i_evt
      integer :: u, i, num_event
      num_event = 0
      if (present (i_evt)) num_event = i_evt
      u = output_unit (unit);  if (u < 0)  return
      write (u, *) num_event, NHEP
      do i = 1, NHEP
         write (u, *) i, ISTHEP(i), IDHEP(i), JMOHEP(:,i), JDAHEP(:,i)
         write (u, *) PHEP(:,i)
         write (u, *) VHEP(1:4,i)
      end do
    end subroutine hepevt_write_athena
```

8.3   HepMC events

This section provides the interface to the HepMC C++ library for handling
Monte-Carlo events.

Each C++ class of HepMC that we use is mirrored by a Fortran type, which
contains as its only component the C pointer to the C++ object.

Each C++ method of HepMC that we use has a C wrapper function. This
function takes a pointer to the host object as its first argument. Further ar-

guments are either C pointers, or in the case of simple types (integer, real), interoperable C/Fortran objects.

The C wrapper functions have explicit interfaces in the Fortran module. They are called by Fortran wrapper procedures. These are treated as methods of the corresponding Fortran type.

⟨hepmc_interface.f90⟩≡
  ⟨*File header*⟩

  module hepmc_interface

    use iso_c_binding !NODEP!
  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
    use constants !NODEP!
    use lorentz !NODEP!
    use models
    use flavors
    use colors
    use helicities
    use quantum_numbers
    use polarizations

  ⟨*Standard module head*⟩

  ⟨*HepMC interface: public*⟩

  ⟨*HepMC interface: types*⟩

  ⟨*HepMC interface: interfaces*⟩

  contains

  ⟨*HepMC interface: procedures*⟩

  end module hepmc_interface

8.3.1 Interface check

This function can be called in order to verify that we are using the actual HepMC library, and not the dummy version.

⟨*HepMC interface: interfaces*⟩≡
    interface
       logical(c_bool) function hepmc_available () bind(C)
         import
       end function hepmc_available
    end interface

⟨*HepMC interface: public*⟩≡
    public :: hepmc_is_available

⟨*HepMC interface: procedures*⟩≡
    function hepmc_is_available () result (flag)
      logical :: flag
      flag = hepmc_available ()

701

```
    end function hepmc_is_available
```

8.3.2 FourVector

The C version of four-vectors is often transferred by value, and the associated
procedures are all inlined. The wrapper needs to transfer by reference, so we
create FourVector objects on the heap which have to be deleted explicitly. The
input is a `vector4_t` or `vector3_t` object from the `lorentz` module.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_four_vector_t
```

⟨*HepMC interface: types*⟩≡
```
    type :: hepmc_four_vector_t
      private
      type(c_ptr) :: obj
    end type hepmc_four_vector_t
```

In the C constructor, the zero-component (fourth argument) is optional; if miss-
ing, it is set to zero. The Fortran version has initializer form and takes either a
three-vector or a four-vector. A further version extracts the four-vector from a
HepMC particle object.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      type(c_ptr) function new_four_vector_xyz (x, y, z) bind(C)
        import
        real(c_double), value :: x, y, z
      end function new_four_vector_xyz
    end interface
    interface
      type(c_ptr) function new_four_vector_xyzt (x, y, z, t) bind(C)
        import
        real(c_double), value :: x, y, z, t
      end function new_four_vector_xyzt
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_four_vector_init
```

⟨*HepMC interface: interfaces*⟩+≡
```
    interface hepmc_four_vector_init
      module procedure hepmc_four_vector_init_v4
      module procedure hepmc_four_vector_init_v3
      module procedure hepmc_four_vector_init_hepmc_prt
    end interface
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_four_vector_init_v4 (pp, p)
      type(hepmc_four_vector_t), intent(out) :: pp
      type(vector4_t), intent(in) :: p
      real(default), dimension(0:3) :: pa
      pa = vector4_get_components (p)
      pp%obj = new_four_vector_xyzt &
           (real (pa(1), c_double), &
```

```
                real (pa(2), c_double), &
                real (pa(3), c_double), &
                real (pa(0), c_double))
      end subroutine hepmc_four_vector_init_v4

      subroutine hepmc_four_vector_init_v3 (pp, p)
        type(hepmc_four_vector_t), intent(out) :: pp
        type(vector3_t), intent(in) :: p
        real(default), dimension(3) :: pa
        pa = vector3_get_components (p)
        pp%obj = new_four_vector_xyz &
              (real (pa(1), c_double), &
               real (pa(2), c_double), &
               real (pa(3), c_double))
      end subroutine hepmc_four_vector_init_v3

      subroutine hepmc_four_vector_init_hepmc_prt (pp, prt)
        type(hepmc_four_vector_t), intent(out) :: pp
        type(hepmc_particle_t), intent(in) :: prt
        pp%obj = gen_particle_momentum (prt%obj)
      end subroutine hepmc_four_vector_init_hepmc_prt
```

Here, the destructor is explicitly needed.

⟨*HepMC interface: interfaces*⟩+≡
```
      interface
         subroutine four_vector_delete (p_obj) bind(C)
           import
           type(c_ptr), value :: p_obj
         end subroutine four_vector_delete
      end interface
```

⟨*HepMC interface: public*⟩+≡
```
      public :: hepmc_four_vector_final
```

⟨*HepMC interface: procedures*⟩+≡
```
      subroutine hepmc_four_vector_final (p)
        type(hepmc_four_vector_t), intent(inout) :: p
        call four_vector_delete (p%obj)
      end subroutine hepmc_four_vector_final
```

Convert to a Lorentz vector.

⟨*HepMC interface: interfaces*⟩+≡
```
      interface
         function four_vector_px (p_obj) result (px) bind(C)
           import
           real(c_double) :: px
           type(c_ptr), value :: p_obj
         end function four_vector_px
      end interface
      interface
         function four_vector_py (p_obj) result (py) bind(C)
           import
           real(c_double) :: py
           type(c_ptr), value :: p_obj
```

```
        end function four_vector_py
      end interface
      interface
        function four_vector_pz (p_obj) result (pz) bind(C)
          import
          real(c_double) :: pz
          type(c_ptr), value :: p_obj
        end function four_vector_pz
      end interface
      interface
        function four_vector_e (p_obj) result (e) bind(C)
          import
          real(c_double) :: e
          type(c_ptr), value :: p_obj
        end function four_vector_e
      end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_four_vector_to_vector4
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_four_vector_to_vector4 (pp, p)
      type(hepmc_four_vector_t), intent(in) :: pp
      type(vector4_t), intent(out) :: p
      real(default) :: E
      real(default), dimension(3) :: p3
      E = four_vector_e (pp%obj)
      p3(1) = four_vector_px (pp%obj)
      p3(2) = four_vector_py (pp%obj)
      p3(3) = four_vector_pz (pp%obj)
      p = vector4_moving (E, vector3_moving (p3))
    end subroutine hepmc_four_vector_to_vector4
```

8.3.3 Polarization

Polarization objects are temporarily used for assigning particle polarization. We add a flag `polarized`. If this is false, the polarization is not set and should not be transferred to `hepmc_particle` objects.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_polarization_t
```

⟨*HepMC interface: types*⟩+≡
```
    type :: hepmc_polarization_t
      private
      logical :: polarized = .false.
      type(c_ptr) :: obj
    end type hepmc_polarization_t
```

Constructor. The C wrapper takes polar and azimuthal angle as arguments. The Fortran version allows for either a complete polarization density matrix, or for a definite (diagonal) helicity.

HepMC does not allow to specify the degree of polarization, therefore we have to map it to either 0 or 1. We choose 0 for polarization less than 0.5 *and 1 for*

*polarization greater than 0.5. Even this simplification works only for spin-1/2
and for massless particles; massive vector bosons cannot be treated this way. In
particular, zero helicity is always translated as unpolarized.*

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   type(c_ptr) function new_polarization (theta, phi) bind(C)
     import
     real(c_double), value :: theta, phi
   end function new_polarization
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_polarization_init
```

⟨*HepMC interface: interfaces*⟩+≡
```
interface hepmc_polarization_init
   module procedure hepmc_polarization_init_pol
   module procedure hepmc_polarization_init_hel
end interface
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_polarization_init_pol (hpol, pol)
  type(hepmc_polarization_t), intent(out) :: hpol
  type(polarization_t), intent(in) :: pol
  real(default) :: r, theta, phi
  if (polarization_is_polarized (pol)) then
     call polarization_to_angles (pol, r, theta, phi)
     if (r >= 0.5) then
        hpol%polarized = .true.
        hpol%obj = new_polarization &
             (real (theta, c_double), real (phi, c_double))
     end if
  end if
end subroutine hepmc_polarization_init_pol

subroutine hepmc_polarization_init_hel (hpol, hel)
  type(hepmc_polarization_t), intent(out) :: hpol
  type(helicity_t), intent(in) :: hel
  integer, dimension(2) :: h
  if (helicity_is_defined (hel)) then
     h = helicity_get (hel)
     select case (h(1))
     case (1:)
        hpol%polarized = .true.
        hpol%obj = new_polarization (0._c_double, 0._c_double)
     case (:-1)
        hpol%polarized = .true.
        hpol%obj = new_polarization (real (pi, c_double), 0._c_double)
     end select
  end if
end subroutine hepmc_polarization_init_hel
```

Destructor. The C object is deallocated only if the `polarized` flag is set.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
```

```
      subroutine polarization_delete (pol_obj) bind(C)
        import
        type(c_ptr), value :: pol_obj
      end subroutine polarization_delete
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_polarization_final
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_polarization_final (hpol)
      type(hepmc_polarization_t), intent(inout) :: hpol
      if (hpol%polarized)  call polarization_delete (hpol%obj)
    end subroutine hepmc_polarization_final
```

Recover polarization from HepMC polarization object (with the abovementioned deficiencies).

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      function polarization_theta (pol_obj) result (theta) bind(C)
        import
        real(c_double) :: theta
        type(c_ptr), value :: pol_obj
      end function polarization_theta
    end interface
    interface
      function polarization_phi (pol_obj) result (phi) bind(C)
        import
        real(c_double) :: phi
        type(c_ptr), value :: pol_obj
      end function polarization_phi
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_polarization_to_pol
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_polarization_to_pol (hpol, flv, pol)
      type(hepmc_polarization_t), intent(in) :: hpol
      type(flavor_t), intent(in) :: flv
      type(polarization_t), intent(out) :: pol
      real(default) :: theta, phi
      theta = polarization_theta (hpol%obj)
      phi = polarization_phi (hpol%obj)
      call polarization_init_angles (pol, flv, 1._default, theta, phi)
    end subroutine hepmc_polarization_to_pol
```

Recover helicity. Here, ϕ is ignored and only the sign of $\cos\theta$ is relevant, mapped to positive/negative helicity.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_polarization_to_hel
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_polarization_to_hel (hpol, flv, hel)
      type(hepmc_polarization_t), intent(in) :: hpol
```

```
      type(flavor_t), intent(in) :: flv
      type(helicity_t), intent(out) :: hel
      real(default) :: theta
      integer :: hmax
      theta = polarization_theta (hpol%obj)
      hmax = flavor_get_spin_type (flv) / 2
      call helicity_init (hel, sign (hmax, nint (cos (theta))))
    end subroutine hepmc_polarization_to_hel
```

8.3.4 GenParticle

Particle objects have the obvious meaning.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_t
```

⟨*HepMC interface: types*⟩+≡
```
    type :: hepmc_particle_t
       private
       type(c_ptr) :: obj
    end type hepmc_particle_t
```

Constructor. The C version takes a FourVector object, which in the Fortran wrapper is created on the fly from a `vector4` Lorentz vector.

No destructor is needed as long as all particles are entered into vertex containers.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       type(c_ptr) function new_gen_particle (prt_obj, pdg_id, status) bind(C)
         import
         type(c_ptr), value :: prt_obj
         integer(c_int), value :: pdg_id, status
       end function new_gen_particle
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_init
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_particle_init (prt, p, pdg, status)
      type(hepmc_particle_t), intent(out) :: prt
      type(vector4_t), intent(in) :: p
      integer, intent(in) :: pdg, status
      type(hepmc_four_vector_t) :: pp
      call hepmc_four_vector_init (pp, p)
      prt%obj = new_gen_particle (pp%obj, int (pdg, c_int), int (status, c_int))
      call hepmc_four_vector_final (pp)
    end subroutine hepmc_particle_init
```

Set the particle color flow.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine gen_particle_set_flow (prt_obj, code_index, code) bind(C)
```

```
       import
       type(c_ptr), value :: prt_obj
       integer(c_int), value :: code_index, code
     end subroutine gen_particle_set_flow
   end interface
```

⟨*HepMC interface: public*⟩+≡
```
   public :: hepmc_particle_set_color
```

⟨*HepMC interface: procedures*⟩+≡
```
   subroutine hepmc_particle_set_color (prt, col)
     type(hepmc_particle_t), intent(inout) :: prt
     type(color_t), intent(in) :: col
     integer(c_int) :: c
     c = color_get_col (col)
     if (c /= 0)  call gen_particle_set_flow (prt%obj, 1_c_int, c)
     c = color_get_acl (col)
     if (c /= 0)  call gen_particle_set_flow (prt%obj, 2_c_int, c)
   end subroutine hepmc_particle_set_color
```

Set the particle polarization. For the restrictions on particle polarization in
HepMC, see above `hepmc_polarization_init`.

⟨*HepMC interface: interfaces*⟩+≡
```
   interface
     subroutine gen_particle_set_polarization (prt_obj, pol_obj) bind(C)
       import
       type(c_ptr), value :: prt_obj, pol_obj
     end subroutine gen_particle_set_polarization
   end interface
```

⟨*HepMC interface: public*⟩+≡
```
   public :: hepmc_particle_set_polarization
```

⟨*HepMC interface: interfaces*⟩+≡
```
   interface hepmc_particle_set_polarization
     module procedure hepmc_particle_set_polarization_pol
     module procedure hepmc_particle_set_polarization_hel
   end interface
```

⟨*HepMC interface: procedures*⟩+≡
```
   subroutine hepmc_particle_set_polarization_pol (prt, pol)
     type(hepmc_particle_t), intent(inout) :: prt
     type(polarization_t), intent(in) :: pol
     type(hepmc_polarization_t) :: hpol
     call hepmc_polarization_init (hpol, pol)
     if (hpol%polarized)  call gen_particle_set_polarization (prt%obj, hpol%obj)
     call hepmc_polarization_final (hpol)
   end subroutine hepmc_particle_set_polarization_pol

   subroutine hepmc_particle_set_polarization_hel (prt, hel)
     type(hepmc_particle_t), intent(inout) :: prt
     type(helicity_t), intent(in) :: hel
     type(hepmc_polarization_t) :: hpol
     call hepmc_polarization_init (hpol, hel)
     if (hpol%polarized)  call gen_particle_set_polarization (prt%obj, hpol%obj)
     call hepmc_polarization_final (hpol)
```

```
    end subroutine hepmc_particle_set_polarization_hel
```

Return the HepMC barcode (unique integer ID) of the particle.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       function gen_particle_barcode (prt_obj) result (barcode) bind(C)
         import
         integer(c_int) :: barcode
         type(c_ptr), value :: prt_obj
       end function gen_particle_barcode
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_get_barcode
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_particle_get_barcode (prt) result (barcode)
      integer :: barcode
      type(hepmc_particle_t), intent(in) :: prt
      barcode = gen_particle_barcode (prt%obj)
    end function hepmc_particle_get_barcode
```

Return the four-vector component of the particle object as a `vector4_t` Lorentz vector.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       type(c_ptr) function gen_particle_momentum (prt_obj) bind(C)
         import
         type(c_ptr), value :: prt_obj
       end function gen_particle_momentum
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_get_momentum
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_particle_get_momentum (prt) result (p)
      type(vector4_t) :: p
      type(hepmc_particle_t), intent(in) :: prt
      type(hepmc_four_vector_t) :: pp
      call hepmc_four_vector_init (pp, prt)
      call hepmc_four_vector_to_vector4 (pp, p)
      call hepmc_four_vector_final (pp)
    end function hepmc_particle_get_momentum
```

Return the invariant mass squared of the particle object. HepMC stores the signed invariant mass (no squaring).

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       function gen_particle_generated_mass (prt_obj) result (mass) bind(C)
         import
         real(c_double) :: mass
         type(c_ptr), value :: prt_obj
       end function gen_particle_generated_mass
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_get_mass_squared
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_particle_get_mass_squared (prt) result (m2)
      real(default) :: m2
      type(hepmc_particle_t), intent(in) :: prt
      real(default) :: m
      m = gen_particle_generated_mass (prt%obj)
      m2 = sign (m**2, m)
    end function hepmc_particle_get_mass_squared
```

Return the PDG ID:

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       function gen_particle_pdg_id (prt_obj) result (pdg_id) bind(C)
         import
         integer(c_int) :: pdg_id
         type(c_ptr), value :: prt_obj
       end function gen_particle_pdg_id
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_get_pdg
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_particle_get_pdg (prt) result (pdg)
      integer :: pdg
      type(hepmc_particle_t), intent(in) :: prt
      pdg = gen_particle_pdg_id (prt%obj)
    end function hepmc_particle_get_pdg
```

Return the status code:

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       function gen_particle_status (prt_obj) result (status) bind(C)
         import
         integer(c_int) :: status
         type(c_ptr), value :: prt_obj
       end function gen_particle_status
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_get_status
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_particle_get_status (prt) result (status)
      integer :: status
      type(hepmc_particle_t), intent(in) :: prt
      status = gen_particle_status (prt%obj)
    end function hepmc_particle_get_status
```

Return the production/decay vertex (as a pointer, no finalization necessary).

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
```

```
      type(c_ptr) function gen_particle_production_vertex (prt_obj) bind(C)
        import
        type(c_ptr), value :: prt_obj
      end function gen_particle_production_vertex
    end interface
    interface
      type(c_ptr) function gen_particle_end_vertex (prt_obj) bind(C)
        import
        type(c_ptr), value :: prt_obj
      end function gen_particle_end_vertex
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_get_production_vertex
    public :: hepmc_particle_get_decay_vertex
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_particle_get_production_vertex (prt) result (v)
      type(hepmc_vertex_t) :: v
      type(hepmc_particle_t), intent(in) :: prt
      v%obj = gen_particle_production_vertex (prt%obj)
    end function hepmc_particle_get_production_vertex

    function hepmc_particle_get_decay_vertex (prt) result (v)
      type(hepmc_vertex_t) :: v
      type(hepmc_particle_t), intent(in) :: prt
      v%obj = gen_particle_end_vertex (prt%obj)
    end function hepmc_particle_get_decay_vertex
```

Return the number of parents/children.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_get_n_parents
    public :: hepmc_particle_get_n_children
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_particle_get_n_parents (prt) result (n_parents)
      integer :: n_parents
      type(hepmc_particle_t), intent(in) :: prt
      type(hepmc_vertex_t) :: v
      v = hepmc_particle_get_production_vertex (prt)
      if (hepmc_vertex_is_valid (v)) then
         n_parents = hepmc_vertex_get_n_in (v)
      else
         n_parents = 0
      end if
    end function hepmc_particle_get_n_parents

    function hepmc_particle_get_n_children (prt) result (n_children)
      integer :: n_children
      type(hepmc_particle_t), intent(in) :: prt
      type(hepmc_vertex_t) :: v
      v = hepmc_particle_get_decay_vertex (prt)
      if (hepmc_vertex_is_valid (v)) then
         n_children = hepmc_vertex_get_n_out (v)
      else
```

```
        n_children = 0
      end if
    end function hepmc_particle_get_n_children
```

Convenience function: Return the array of parent particles for a given HepMC particle. The contents are HepMC barcodes that still have to be mapped to the particle indices.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_get_parent_barcodes
    public :: hepmc_particle_get_child_barcodes
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_particle_get_parent_barcodes (prt) result (parent_barcode)
      type(hepmc_particle_t), intent(in) :: prt
      integer, dimension(:), allocatable :: parent_barcode
      type(hepmc_vertex_t) :: v
      type(hepmc_vertex_particle_in_iterator_t) :: it
      integer :: i
      v = hepmc_particle_get_production_vertex (prt)
      if (hepmc_vertex_is_valid (v)) then
         allocate (parent_barcode (hepmc_vertex_get_n_in (v)))
         if (size (parent_barcode) /= 0) then
            call hepmc_vertex_particle_in_iterator_init (it, v)
            do i = 1, size (parent_barcode)
               parent_barcode(i) = hepmc_particle_get_barcode &
                    (hepmc_vertex_particle_in_iterator_get (it))
               call hepmc_vertex_particle_in_iterator_advance (it)
            end do
            call hepmc_vertex_particle_in_iterator_final (it)
         end if
      else
         allocate (parent_barcode (0))
      end if
    end function hepmc_particle_get_parent_barcodes

    function hepmc_particle_get_child_barcodes (prt) result (child_barcode)
      type(hepmc_particle_t), intent(in) :: prt
      integer, dimension(:), allocatable :: child_barcode
      type(hepmc_vertex_t) :: v
      type(hepmc_vertex_particle_out_iterator_t) :: it
      integer :: i
      v = hepmc_particle_get_decay_vertex (prt)
      if (hepmc_vertex_is_valid (v)) then
         allocate (child_barcode (hepmc_vertex_get_n_out (v)))
         call hepmc_vertex_particle_out_iterator_init (it, v)
         if (size (child_barcode) /= 0) then
            do i = 1, size (child_barcode)
               child_barcode(i) = hepmc_particle_get_barcode &
                    (hepmc_vertex_particle_out_iterator_get (it))
               call hepmc_vertex_particle_out_iterator_advance (it)
            end do
            call hepmc_vertex_particle_out_iterator_final (it)
         end if
      else
```

```
        allocate (child_barcode (0))
      end if
    end function hepmc_particle_get_child_barcodes
```

Return the polarization (assuming that the particle is completely polarized).
Note that the generated polarization object needs finalization.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       type(c_ptr) function gen_particle_polarization (prt_obj) bind(C)
         import
         type(c_ptr), value :: prt_obj
       end function gen_particle_polarization
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_get_polarization
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_particle_get_polarization (prt) result (pol)
      type(hepmc_polarization_t) :: pol
      type(hepmc_particle_t), intent(in) :: prt
      pol%obj = gen_particle_polarization (prt%obj)
    end function hepmc_particle_get_polarization
```

Return the particle color as a two-dimensional array (color, anticolor).

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       function gen_particle_flow (prt_obj, code_index) result (code) bind(C)
         import
         integer(c_int) :: code
         type(c_ptr), value :: prt_obj
         integer(c_int), value :: code_index
       end function gen_particle_flow
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_get_color
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_particle_get_color (prt) result (col)
      integer, dimension(2) :: col
      type(hepmc_particle_t), intent(in) :: prt
      col(1) = gen_particle_flow (prt%obj, 1)
      col(2) = - gen_particle_flow (prt%obj, 2)
    end function hepmc_particle_get_color
```

8.3.5   GenVertex

Vertices are made of particles (incoming and outgoing).

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_t
```

⟨*HepMC interface: types*⟩+≡
```
type :: hepmc_vertex_t
   private
   type(c_ptr) :: obj
end type hepmc_vertex_t
```

Constructor. Two versions, one plain, one with the position in space and time (measured in mm) as argument. The Fortran version has initializer form, and the vertex position is an optional argument.

A destructor is unnecessary as long as all vertices are entered into an event container.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   type(c_ptr) function new_gen_vertex () bind(C)
     import
   end function new_gen_vertex
end interface
interface
   type(c_ptr) function new_gen_vertex_pos (prt_obj) bind(C)
     import
     type(c_ptr), value :: prt_obj
   end function new_gen_vertex_pos
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_vertex_init
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_vertex_init (v, x)
  type(hepmc_vertex_t), intent(out) :: v
  type(vector4_t), intent(in), optional :: x
  type(hepmc_four_vector_t) :: pos
  if (present (x)) then
     call hepmc_four_vector_init (pos, x)
     v%obj = new_gen_vertex_pos (pos%obj)
     call hepmc_four_vector_final (pos)
  else
     v%obj = new_gen_vertex ()
  end if
end subroutine hepmc_vertex_init
```

Return true if the vertex pointer is non-null:

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   function gen_vertex_is_valid (v_obj) result (flag) bind(C)
     import
     logical(c_bool) :: flag
     type(c_ptr), value :: v_obj
   end function gen_vertex_is_valid
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_vertex_is_valid
```

714

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_vertex_is_valid (v) result (flag)
      logical :: flag
      type(hepmc_vertex_t), intent(in) :: v
      flag = gen_vertex_is_valid (v%obj)
    end function hepmc_vertex_is_valid
```

Add a particle to a vertex, incoming or outgoing.
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      subroutine gen_vertex_add_particle_in (v_obj, prt_obj) bind(C)
        import
        type(c_ptr), value :: v_obj, prt_obj
      end subroutine gen_vertex_add_particle_in
    end interface
    interface
      subroutine gen_vertex_add_particle_out (v_obj, prt_obj) bind(C)
        import
        type(c_ptr), value :: v_obj, prt_obj
      end subroutine gen_vertex_add_particle_out
    end interface
```
⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_add_particle_in
    public :: hepmc_vertex_add_particle_out
```
⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_add_particle_in (v, prt)
      type(hepmc_vertex_t), intent(inout) :: v
      type(hepmc_particle_t), intent(in) :: prt
      call gen_vertex_add_particle_in (v%obj, prt%obj)
    end subroutine hepmc_vertex_add_particle_in

    subroutine hepmc_vertex_add_particle_out (v, prt)
      type(hepmc_vertex_t), intent(inout) :: v
      type(hepmc_particle_t), intent(in) :: prt
      call gen_vertex_add_particle_out (v%obj, prt%obj)
    end subroutine hepmc_vertex_add_particle_out
```

Return the number of incoming/outgoing particles.
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      function gen_vertex_particles_in_size (v_obj) result (size) bind(C)
        import
        integer(c_int) :: size
        type(c_ptr), value :: v_obj
      end function gen_vertex_particles_in_size
    end interface
    interface
      function gen_vertex_particles_out_size (v_obj) result (size) bind(C)
        import
        integer(c_int) :: size
        type(c_ptr), value :: v_obj
      end function gen_vertex_particles_out_size
```

```
        end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_get_n_in
    public :: hepmc_vertex_get_n_out
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_vertex_get_n_in (v) result (n_in)
      integer :: n_in
      type(hepmc_vertex_t), intent(in) :: v
      n_in = gen_vertex_particles_in_size (v%obj)
    end function hepmc_vertex_get_n_in

    function hepmc_vertex_get_n_out (v) result (n_out)
      integer :: n_out
      type(hepmc_vertex_t), intent(in) :: v
      n_out = gen_vertex_particles_out_size (v%obj)
    end function hepmc_vertex_get_n_out
```

8.3.6   Vertex-particle-in iterator

This iterator iterates over all incoming particles in an vertex. We store a pointer
to the vertex in addition to the iterator. This allows for simple end checking.

The iterator is actually a constant iterator; it can only read.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_in_iterator_t
```

⟨*HepMC interface: types*⟩+≡
```
    type :: hepmc_vertex_particle_in_iterator_t
      private
      type(c_ptr) :: obj
      type(c_ptr) :: v_obj
    end type hepmc_vertex_particle_in_iterator_t
```

Constructor. The iterator is initialized at the first particle in the vertex.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      type(c_ptr) function &
          new_vertex_particles_in_const_iterator (v_obj) bind(C)
        import
        type(c_ptr), value :: v_obj
      end function new_vertex_particles_in_const_iterator
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_in_iterator_init
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_in_iterator_init (it, v)
      type(hepmc_vertex_particle_in_iterator_t), intent(out) :: it
      type(hepmc_vertex_t), intent(in) :: v
      it%obj = new_vertex_particles_in_const_iterator (v%obj)
      it%v_obj = v%obj
    end subroutine hepmc_vertex_particle_in_iterator_init
```

Destructor. Necessary because the iterator is allocated on the heap.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine vertex_particles_in_const_iterator_delete (it_obj) bind(C)
         import
         type(c_ptr), value :: it_obj
       end subroutine vertex_particles_in_const_iterator_delete
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_in_iterator_final
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_in_iterator_final (it)
      type(hepmc_vertex_particle_in_iterator_t), intent(inout) :: it
      call vertex_particles_in_const_iterator_delete (it%obj)
    end subroutine hepmc_vertex_particle_in_iterator_final
```


Increment

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine vertex_particles_in_const_iterator_advance (it_obj) bind(C)
         import
         type(c_ptr), value :: it_obj
       end subroutine vertex_particles_in_const_iterator_advance
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_in_iterator_advance
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_in_iterator_advance (it)
      type(hepmc_vertex_particle_in_iterator_t), intent(inout) :: it
      call vertex_particles_in_const_iterator_advance (it%obj)
    end subroutine hepmc_vertex_particle_in_iterator_advance
```


Reset to the beginning

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine vertex_particles_in_const_iterator_reset &
            (it_obj, v_obj) bind(C)
         import
         type(c_ptr), value :: it_obj, v_obj
       end subroutine vertex_particles_in_const_iterator_reset
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_in_iterator_reset
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_in_iterator_reset (it)
      type(hepmc_vertex_particle_in_iterator_t), intent(inout) :: it
      call vertex_particles_in_const_iterator_reset (it%obj, it%v_obj)
    end subroutine hepmc_vertex_particle_in_iterator_reset
```

Test: return true as long as we are not past the end.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
  function vertex_particles_in_const_iterator_is_valid &
      (it_obj, v_obj) result (flag) bind(C)
    import
    logical(c_bool) :: flag
    type(c_ptr), value :: it_obj, v_obj
  end function vertex_particles_in_const_iterator_is_valid
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_vertex_particle_in_iterator_is_valid
```

⟨*HepMC interface: procedures*⟩+≡
```
function hepmc_vertex_particle_in_iterator_is_valid (it) result (flag)
  logical :: flag
  type(hepmc_vertex_particle_in_iterator_t), intent(in) :: it
  flag = vertex_particles_in_const_iterator_is_valid (it%obj, it%v_obj)
end function hepmc_vertex_particle_in_iterator_is_valid
```

Return the particle pointed to by the iterator. (The particle object should not be finalized, since it contains merely a pointer to the particle which is owned by the vertex.)

⟨*HepMC interface: interfaces*⟩+≡
```
interface
  type(c_ptr) function &
      vertex_particles_in_const_iterator_get (it_obj) bind(C)
    import
    type(c_ptr), value :: it_obj
  end function vertex_particles_in_const_iterator_get
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_vertex_particle_in_iterator_get
```

⟨*HepMC interface: procedures*⟩+≡
```
function hepmc_vertex_particle_in_iterator_get (it) result (prt)
  type(hepmc_particle_t) :: prt
  type(hepmc_vertex_particle_in_iterator_t), intent(in) :: it
  prt%obj = vertex_particles_in_const_iterator_get (it%obj)
end function hepmc_vertex_particle_in_iterator_get
```

8.3.7 Vertex-particle-out iterator

This iterator iterates over all incoming particles in an vertex. We store a pointer to the vertex in addition to the iterator. This allows for simple end checking.

The iterator is actually a constant iterator; it can only read.

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_vertex_particle_out_iterator_t
```

⟨*HepMC interface: types*⟩+≡
```
    type :: hepmc_vertex_particle_out_iterator_t
       private
       type(c_ptr) :: obj
       type(c_ptr) :: v_obj
    end type hepmc_vertex_particle_out_iterator_t
```

Constructor. The iterator is initialized at the first particle in the vertex.
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       type(c_ptr) function &
            new_vertex_particles_out_const_iterator (v_obj) bind(C)
         import
         type(c_ptr), value :: v_obj
       end function new_vertex_particles_out_const_iterator
    end interface
```
⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_out_iterator_init
```
⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_out_iterator_init (it, v)
      type(hepmc_vertex_particle_out_iterator_t), intent(out) :: it
      type(hepmc_vertex_t), intent(in) :: v
      it%obj = new_vertex_particles_out_const_iterator (v%obj)
      it%v_obj = v%obj
    end subroutine hepmc_vertex_particle_out_iterator_init
```

Destructor. Necessary because the iterator is allocated on the heap.
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine vertex_particles_out_const_iterator_delete (it_obj) bind(C)
         import
         type(c_ptr), value :: it_obj
       end subroutine vertex_particles_out_const_iterator_delete
    end interface
```
⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_out_iterator_final
```
⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_out_iterator_final (it)
      type(hepmc_vertex_particle_out_iterator_t), intent(inout) :: it
      call vertex_particles_out_const_iterator_delete (it%obj)
    end subroutine hepmc_vertex_particle_out_iterator_final
```

Increment
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine vertex_particles_out_const_iterator_advance (it_obj) bind(C)
         import
         type(c_ptr), value :: it_obj
       end subroutine vertex_particles_out_const_iterator_advance
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_out_iterator_advance
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_out_iterator_advance (it)
      type(hepmc_vertex_particle_out_iterator_t), intent(inout) :: it
      call vertex_particles_out_const_iterator_advance (it%obj)
    end subroutine hepmc_vertex_particle_out_iterator_advance
```

Reset to the beginning

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine vertex_particles_out_const_iterator_reset &
            (it_obj, v_obj) bind(C)
         import
         type(c_ptr), value :: it_obj, v_obj
       end subroutine vertex_particles_out_const_iterator_reset
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_out_iterator_reset
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_out_iterator_reset (it)
      type(hepmc_vertex_particle_out_iterator_t), intent(inout) :: it
      call vertex_particles_out_const_iterator_reset (it%obj, it%v_obj)
    end subroutine hepmc_vertex_particle_out_iterator_reset
```

Test: return true as long as we are not past the end.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       function vertex_particles_out_const_iterator_is_valid &
            (it_obj, v_obj) result (flag) bind(C)
         import
         logical(c_bool) :: flag
         type(c_ptr), value :: it_obj, v_obj
       end function vertex_particles_out_const_iterator_is_valid
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_out_iterator_is_valid
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_vertex_particle_out_iterator_is_valid (it) result (flag)
      logical :: flag
      type(hepmc_vertex_particle_out_iterator_t), intent(in) :: it
      flag = vertex_particles_out_const_iterator_is_valid (it%obj, it%v_obj)
    end function hepmc_vertex_particle_out_iterator_is_valid
```

Return the particle pointed to by the iterator. (The particle object should not be finalized, since it contains merely a pointer to the particle which is owned by the vertex.)

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
```

```
      type(c_ptr) function &
           vertex_particles_out_const_iterator_get (it_obj) bind(C)
        import
        type(c_ptr), value :: it_obj
      end function vertex_particles_out_const_iterator_get
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_out_iterator_get
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_vertex_particle_out_iterator_get (it) result (prt)
      type(hepmc_particle_t) :: prt
      type(hepmc_vertex_particle_out_iterator_t), intent(in) :: it
      prt%obj = vertex_particles_out_const_iterator_get (it%obj)
    end function hepmc_vertex_particle_out_iterator_get
```

8.3.8 GenEvent

The main object of HepMC is a GenEvent. The object is filled by GenVertex objects, which in turn contain GenParticle objects.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_t
```

⟨*HepMC interface: types*⟩+≡
```
    type :: hepmc_event_t
       private
       type(c_ptr) :: obj
    end type hepmc_event_t
```

Constructor. Arguments are process ID (integer) and event ID (integer).
The Fortran version has initializer form.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       type(c_ptr) function new_gen_event (proc_id, event_id) bind(C)
         import
         integer(c_int), value :: proc_id, event_id
       end function new_gen_event
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_init
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_init (evt, proc_id, event_id)
      type(hepmc_event_t), intent(out) :: evt
      integer, intent(in), optional :: proc_id, event_id
      integer(c_int) :: pid, eid
      pid = 0;  if (present (proc_id))  pid = proc_id
      eid = 0;  if (present (event_id)) eid = event_id
      evt%obj = new_gen_event (pid, eid)
    end subroutine hepmc_event_init
```

Destructor.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   subroutine gen_event_delete (evt_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
   end subroutine gen_event_delete
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_event_final
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_event_final (evt)
  type(hepmc_event_t), intent(inout) :: evt
  call gen_event_delete (evt%obj)
end subroutine hepmc_event_final
```

Screen output. Printing to file is possible in principle (using a C++ output
channel), by allowing an argument. Printing to an open Fortran unit is obviously
not possible.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   subroutine gen_event_print (evt_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
   end subroutine gen_event_print
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_event_print
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_event_print (evt)
  type(hepmc_event_t), intent(in) :: evt
  call gen_event_print (evt%obj)
end subroutine hepmc_event_print
```

Add a vertex to the event container.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   subroutine gen_event_add_vertex (evt_obj, v_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
     type(c_ptr), value :: v_obj
   end subroutine gen_event_add_vertex
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_event_add_vertex
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_event_add_vertex (evt, v)
  type(hepmc_event_t), intent(inout) :: evt
  type(hepmc_vertex_t), intent(in) :: v
  call gen_event_add_vertex (evt%obj, v%obj)
```

```
    end subroutine hepmc_event_add_vertex
```

Mark a particular vertex as the signal process (hard interaction).

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine gen_event_set_signal_process_vertex (evt_obj, v_obj) bind(C)
         import
         type(c_ptr), value :: evt_obj
         type(c_ptr), value :: v_obj
       end subroutine gen_event_set_signal_process_vertex
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_set_signal_process_vertex
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_set_signal_process_vertex (evt, v)
      type(hepmc_event_t), intent(inout) :: evt
      type(hepmc_vertex_t), intent(in) :: v
      call gen_event_set_signal_process_vertex (evt%obj, v%obj)
    end subroutine hepmc_event_set_signal_process_vertex
```

8.3.9 Event-particle iterator

This iterator iterates over all particles in an event. We store a pointer to the event in addition to the iterator. This allows for simple end checking.

The iterator is actually a constant iterator; it can only read.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_particle_iterator_t
```

⟨*HepMC interface: types*⟩+≡
```
    type :: hepmc_event_particle_iterator_t
      private
      type(c_ptr) :: obj
      type(c_ptr) :: evt_obj
    end type hepmc_event_particle_iterator_t
```

Constructor. The iterator is initialized at the first particle in the event.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       type(c_ptr) function new_event_particle_const_iterator (evt_obj) bind(C)
         import
         type(c_ptr), value :: evt_obj
       end function new_event_particle_const_iterator
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_particle_iterator_init
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_particle_iterator_init (it, evt)
      type(hepmc_event_particle_iterator_t), intent(out) :: it
      type(hepmc_event_t), intent(in) :: evt
```

```
      it%obj = new_event_particle_const_iterator (evt%obj)
      it%evt_obj = evt%obj
    end subroutine hepmc_event_particle_iterator_init
```

Destructor. Necessary because the iterator is allocated on the heap.
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      subroutine event_particle_const_iterator_delete (it_obj) bind(C)
        import
        type(c_ptr), value :: it_obj
      end subroutine event_particle_const_iterator_delete
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_particle_iterator_final
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_particle_iterator_final (it)
      type(hepmc_event_particle_iterator_t), intent(inout) :: it
      call event_particle_const_iterator_delete (it%obj)
    end subroutine hepmc_event_particle_iterator_final
```

Increment
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      subroutine event_particle_const_iterator_advance (it_obj) bind(C)
        import
        type(c_ptr), value :: it_obj
      end subroutine event_particle_const_iterator_advance
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_particle_iterator_advance
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_particle_iterator_advance (it)
      type(hepmc_event_particle_iterator_t), intent(inout) :: it
      call event_particle_const_iterator_advance (it%obj)
    end subroutine hepmc_event_particle_iterator_advance
```

Reset to the beginning
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      subroutine event_particle_const_iterator_reset (it_obj, evt_obj) bind(C)
        import
        type(c_ptr), value :: it_obj, evt_obj
      end subroutine event_particle_const_iterator_reset
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_particle_iterator_reset
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_particle_iterator_reset (it)
      type(hepmc_event_particle_iterator_t), intent(inout) :: it
      call event_particle_const_iterator_reset (it%obj, it%evt_obj)
    end subroutine hepmc_event_particle_iterator_reset
```

Test: return true as long as we are not past the end.
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      function event_particle_const_iterator_is_valid &
          (it_obj, evt_obj) result (flag) bind(C)
        import
        logical(c_bool) :: flag
        type(c_ptr), value :: it_obj, evt_obj
      end function event_particle_const_iterator_is_valid
    end interface
```
⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_particle_iterator_is_valid
```
⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_event_particle_iterator_is_valid (it) result (flag)
      logical :: flag
      type(hepmc_event_particle_iterator_t), intent(in) :: it
      flag = event_particle_const_iterator_is_valid (it%obj, it%evt_obj)
    end function hepmc_event_particle_iterator_is_valid
```

Return the particle pointed to by the iterator. (The particle object should not
be finalized, since it contains merely a pointer to the particle which is owned by
the vertex.)
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      type(c_ptr) function event_particle_const_iterator_get (it_obj) bind(C)
        import
        type(c_ptr), value :: it_obj
      end function event_particle_const_iterator_get
    end interface
```
⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_particle_iterator_get
```
⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_event_particle_iterator_get (it) result (prt)
      type(hepmc_particle_t) :: prt
      type(hepmc_event_particle_iterator_t), intent(in) :: it
      prt%obj = event_particle_const_iterator_get (it%obj)
    end function hepmc_event_particle_iterator_get
```

8.3.10 I/O streams

There is a specific I/O stream type for handling the output of GenEvent ob-
jects (i.e., Monte Carlo event samples) to file. Opening the file is done by the
constructor, closing by the destructor.
⟨*HepMC interface: public*⟩+≡

```
      public :: hepmc_iostream_t
```

⟨*HepMC interface: types*⟩+≡
```
    type :: hepmc_iostream_t
       private
       type(c_ptr) :: obj
    end type hepmc_iostream_t
```

Constructor for an output stream associated to a file.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       type(c_ptr) function new_io_gen_event_out (filename) bind(C)
         import
         character(c_char), dimension(*), intent(in) :: filename
       end function new_io_gen_event_out
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_iostream_open_out
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_iostream_open_out (iostream, filename)
      type(hepmc_iostream_t), intent(out) :: iostream
      type(string_t), intent(in) :: filename
      iostream%obj = new_io_gen_event_out (char (filename) // c_null_char)
    end subroutine hepmc_iostream_open_out
```

Constructor for an input stream associated to a file.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       type(c_ptr) function new_io_gen_event_in (filename) bind(C)
         import
         character(c_char), dimension(*), intent(in) :: filename
       end function new_io_gen_event_in
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_iostream_open_in
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_iostream_open_in (iostream, filename)
      type(hepmc_iostream_t), intent(out) :: iostream
      type(string_t), intent(in) :: filename
      iostream%obj = new_io_gen_event_in (char (filename) // c_null_char)
    end subroutine hepmc_iostream_open_in
```

Destructor:

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine io_gen_event_delete (io_obj) bind(C)
         import
         type(c_ptr), value :: io_obj
       end subroutine io_gen_event_delete
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_iostream_close
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_iostream_close (iostream)
      type(hepmc_iostream_t), intent(inout) :: iostream
      call io_gen_event_delete (iostream%obj)
    end subroutine hepmc_iostream_close
```

Write a single event to the I/O stream.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      subroutine io_gen_event_write_event (io_obj, evt_obj) bind(C)
        import
        type(c_ptr), value :: io_obj, evt_obj
      end subroutine io_gen_event_write_event
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_iostream_write_event
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_iostream_write_event (iostream, evt)
      type(hepmc_iostream_t), intent(inout) :: iostream
      type(hepmc_event_t), intent(in) :: evt
      call io_gen_event_write_event (iostream%obj, evt%obj)
    end subroutine hepmc_iostream_write_event
```

Read a single event from the I/O stream.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      subroutine io_gen_event_read_event (io_obj, evt_obj) bind(C)
        import
        type(c_ptr), value :: io_obj, evt_obj
      end subroutine io_gen_event_read_event
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_iostream_read_event
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_iostream_read_event (iostream, evt)
      type(hepmc_iostream_t), intent(inout) :: iostream
      type(hepmc_event_t), intent(in) :: evt
      call io_gen_event_read_event (iostream%obj, evt%obj)
    end subroutine hepmc_iostream_read_event
```

8.3.11 Test

This test example is an abridged version from the build-from-scratch example in the HepMC distribution. We create two vertices for $p \to q$ PDF splitting, then a vertex for a $qq \to W^- g$ hard-interaction process, and finally a vertex for $W^- \to qq$ decay. The setup is for LHC kinematics.

Extending the original example, we set color flow for the incoming quarks and polarization for the outgoing photon. For the latter, we have to define a particle-data object for the photon, so a flavor object can be correctly initialized.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_test
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_test
      type(hepmc_event_t) :: evt
      type(hepmc_vertex_t) :: v1, v2, v3, v4
      type(hepmc_particle_t) :: prt1, prt2, prt3, prt4, prt5, prt6, prt7, prt8
      type(hepmc_iostream_t) :: iostream
      type(flavor_t) :: flv
      type(color_t) :: col
      type(polarization_t) :: pol
      type(particle_data_t), target :: photon_data

      ! Initialize a photon flavor object and some polarization
      call particle_data_init (photon_data, var_str ("PHOTON"), 22)
      call particle_data_set (photon_data, spin_type=VECTOR)
      call particle_data_freeze (photon_data)
      call flavor_init (flv, photon_data)
      call polarization_init_angles &
           (pol, flv, 0.6_default, 1._default, 0.5_default)

      ! Event initialization
      call hepmc_event_init (evt, 20, 1)

      ! $p\to q$ splittings
      call hepmc_vertex_init (v1)
      call hepmc_event_add_vertex (evt, v1)
      call hepmc_vertex_init (v2)
      call hepmc_event_add_vertex (evt, v2)
      call particle_init (prt1, &
           0._default, 0._default, 7000._default, 7000._default, &
           2212, 3)
      call hepmc_vertex_add_particle_in (v1, prt1)
      call particle_init (prt2, &
           0._default, 0._default,-7000._default, 7000._default, &
           2212, 3)
      call hepmc_vertex_add_particle_in (v2, prt2)
      call particle_init (prt3, &
           .750_default, -1.569_default, 32.191_default, 32.238_default, &
           1, 3)
      call color_init_from_array (col, (/501/))
      call hepmc_particle_set_color (prt3, col)
      call hepmc_vertex_add_particle_out (v1, prt3)
      call particle_init (prt4, &
           -3.047_default, -19._default, -54.629_default, 57.920_default, &
           -2, 3)
      call color_init_from_array (col, (/-501/))
      call hepmc_particle_set_color (prt4, col)
      call hepmc_vertex_add_particle_out (v2, prt4)
```

```
      ! Hard interaction
      call hepmc_vertex_init (v3)
      call hepmc_event_add_vertex (evt, v3)
      call hepmc_vertex_add_particle_in (v3, prt3)
      call hepmc_vertex_add_particle_in (v3, prt4)
      call particle_init (prt6, &
           -3.813_default, 0.113_default, -1.833_default, 4.233_default, &
           22, 1)
      call hepmc_particle_set_polarization (prt6, pol)
      call hepmc_vertex_add_particle_out (v3, prt6)
      call particle_init (prt5, &
           1.517_default, -20.68_default, -20.605_default, 85.925_default, &
           -24, 3)
      call hepmc_vertex_add_particle_out (v3, prt5)
      call hepmc_event_set_signal_process_vertex (evt, v3)

      ! $W^-$ decay
      call vertex_init_pos (v4, &
           0.12_default, -0.3_default, 0.05_default, 0.004_default)
      call hepmc_event_add_vertex (evt, v4)
      call hepmc_vertex_add_particle_in (v4, prt5)
      call particle_init (prt7, &
           -2.445_default, 28.816_default, 6.082_default, 29.552_default, &
           1, 1)
      call hepmc_vertex_add_particle_out (v4, prt7)
      call particle_init (prt8, &
           3.962_default, -49.498_default, -26.687_default, 56.373_default, &
           -2, 1)
      call hepmc_vertex_add_particle_out (v4, prt8)

      ! Event output
      call hepmc_event_print (evt)
      print *, "Writing to file 'hepmc_test.hepmc.dat'"
      call hepmc_iostream_open_out (iostream , var_str ("hepmc_test.hepmc.dat"))
      call hepmc_iostream_write_event (iostream, evt)
      call hepmc_iostream_close (iostream)
      print *, "Write completed"

      ! Wrapup
      call polarization_final (pol)
      call hepmc_event_final (evt)

    contains

      subroutine vertex_init_pos (v, x, y, z, t)
        type(hepmc_vertex_t), intent(out) :: v
        real(default), intent(in) :: x, y, z, t
        type(vector4_t) :: xx
        xx = vector4_moving (t, vector3_moving ((/x, y, z/)))
        call hepmc_vertex_init (v, xx)
      end subroutine vertex_init_pos

      subroutine particle_init (prt, px, py, pz, E, pdg, status)
        type(hepmc_particle_t), intent(out) :: prt
```

729

```
      real(default), intent(in) :: px, py, pz, E
      integer, intent(in) :: pdg, status
      type(vector4_t) :: p
      p = vector4_moving (E, vector3_moving ((/px, py, pz/)))
      call hepmc_particle_init (prt, p, pdg, status)
    end subroutine particle_init

  end subroutine hepmc_test
```

8.4   Particles

This module defines the `particle_t` object type, and the methods and operations that deal with it.

⟨particles.f90⟩≡
  ⟨*File header*⟩

```
  module particles
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
    use lorentz !NODEP!
    use prt_lists
    use expressions
    use models
    use flavors
    use colors
    use helicities
    use quantum_numbers
    use state_matrices
    use interactions
    use evaluators
    use polarizations
    use event_formats
    use hepmc_interface
```

  ⟨*Standard module head*⟩

  ⟨*Particles: public*⟩

  ⟨*Particles: parameters*⟩

  ⟨*Particles: types*⟩

  ⟨*Particles: interfaces*⟩

```
  contains
```

  ⟨*Particles: procedures*⟩

```
  end module particles
```

8.4.1 The particle type

Particle status codes

The overall status codes (incoming/outgoing etc.) are inherited from the module `prt_lists`.

Polarization status:

⟨*Particles: parameters*⟩≡

```
integer, parameter :: PRT_UNPOLARIZED = 0
integer, parameter :: PRT_DEFINITE_HELICITY = 1
integer, parameter :: PRT_GENERIC_POLARIZATION = 2
```

Definition

The quantum numbers are flavor (from which invariant particle properties can be derived), color, and polarization. The particle may be unpolarized. In this case, `hel` and `pol` are unspecified. If it has a definite helicity, the `hel` component is defined. If it has a generic polarization, the `pol` component is defined. For each particle we store the four-momentum and the invariant mass squared, i.e., the squared norm of the four-momentum. There is also an optional list of parent and child particles, for bookkeeping in physical events.

⟨*Particles: types*⟩≡

```
type :: particle_t
   private
   integer :: status = PRT_UNDEFINED
   integer :: polarization = PRT_UNPOLARIZED
   type(flavor_t) :: flv
   type(color_t) :: col
   type(helicity_t) :: hel
   type(polarization_t) :: pol
   type(vector4_t) :: p = vector4_null
   real(default) :: p2 = 0
   integer, dimension(:), allocatable :: parent
   integer, dimension(:), allocatable :: child
end type particle_t
```

Particle initializers:

⟨*Particles: interfaces*⟩≡

```
interface particle_init
   module procedure particle_init_particle
   module procedure particle_init_state
   module procedure particle_init_hepmc
end interface
```

Copy a particle. (Deep copy) The excludes the parent-child relations.

⟨*Particles: procedures*⟩≡

```
subroutine particle_init_particle (prt_out, prt_in)
  type(particle_t), intent(out) :: prt_out
  type(particle_t), intent(in) :: prt_in
  prt_out%status = prt_in%status
  prt_out%polarization = prt_in%polarization
```

```
      prt_out%flv = prt_in%flv
      prt_out%col = prt_in%col
      prt_out%hel = prt_in%hel
      prt_out%pol = prt_in%pol
      prt_out%p = prt_in%p
      prt_out%p2 = prt_in%p2
    end subroutine particle_init_particle
```

Initialize a particle using a single-particle state matrix which determines flavor, color, and polarization. The state matrix must have unique flavor and color. The factorization mode determines whether the particle is unpolarized, has definite helicity, or generic polarization. This mode is translated into the polarization status.

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_init_state (prt, state, status, mode)
      type(particle_t), intent(out) :: prt
      type(state_matrix_t), intent(in) :: state
      integer, intent(in) :: status, mode
      type(state_iterator_t) :: it
      prt%status = status
      call state_iterator_init (it, state)
      prt%flv = state_iterator_get_flavor (it, 1)
      if (flavor_is_beam_remnant (prt%flv))  prt%status = PRT_BEAM_REMNANT
      prt%col = state_iterator_get_color (it, 1)
      select case (mode)
      case (FM_SELECT_HELICITY)
         prt%hel = state_iterator_get_helicity (it, 1)
         prt%polarization = PRT_DEFINITE_HELICITY
      case (FM_FACTOR_HELICITY)
         call polarization_init_state_matrix (prt%pol, state)
         prt%polarization = PRT_GENERIC_POLARIZATION
      end select
    end subroutine particle_init_state
```

Initialize a particle from a HepMC particle object. The model is necessary for making a fully qualified flavor component. We have the additional flag `polarized` which tells whether the polarization information should be interpreted or ignored, and the lookup array of barcodes. Note that the lookup array is searched linearly, a possible bottleneck for large particle arrays. If necessary, the barcode array could be replaced by a hash table.

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_init_hepmc (prt, hprt, model, polarization, barcode)
      type(particle_t), intent(out) :: prt
      type(hepmc_particle_t), intent(in) :: hprt
      type(model_t), intent(in), target :: model
      integer, intent(in) :: polarization
      integer, dimension(:), intent(in) :: barcode
      type(hepmc_polarization_t) :: hpol
      integer :: n_parents, n_children
      integer, dimension(:), allocatable :: parent_barcode, child_barcode
      integer :: i
      select case (hepmc_particle_get_status (hprt))
```

732

```
            case (1);  prt%status = PRT_OUTGOING
            case (2);  prt%status = PRT_RESONANT
            case (3);  prt%status = PRT_VIRTUAL
            end select
            call flavor_init (prt%flv, hepmc_particle_get_pdg (hprt), model)
            if (flavor_is_beam_remnant (prt%flv))  prt%status = PRT_BEAM_REMNANT
            call color_init (prt%col, hepmc_particle_get_color (hprt))
            prt%polarization = polarization
            select case (polarization)
            case (PRT_DEFINITE_HELICITY)
               hpol = hepmc_particle_get_polarization (hprt)
               call hepmc_polarization_to_hel (hpol, prt%flv, prt%hel)
               call hepmc_polarization_final (hpol)
            case (PRT_GENERIC_POLARIZATION)
               hpol = hepmc_particle_get_polarization (hprt)
               call hepmc_polarization_to_pol (hpol, prt%flv, prt%pol)
               call hepmc_polarization_final (hpol)
            end select
            prt%p = hepmc_particle_get_momentum (hprt)
            prt%p2 = hepmc_particle_get_mass_squared (hprt)
            n_parents  = hepmc_particle_get_n_parents  (hprt)
            n_children = hepmc_particle_get_n_children (hprt)
            allocate (parent_barcode (n_parents),  prt%parent (n_parents))
            allocate (child_barcode  (n_children), prt%child  (n_children))
            parent_barcode = hepmc_particle_get_parent_barcodes (hprt)
            child_barcode  = hepmc_particle_get_child_barcodes  (hprt)
            do i = 1, size (barcode)
               where (parent_barcode == barcode(i))  prt%parent = i
               where (child_barcode  == barcode(i))  prt%child  = i
            end do
            if (prt%status == PRT_VIRTUAL .and. n_parents == 0) &
                 prt%status = PRT_INCOMING
          end subroutine particle_init_hepmc
```

Finalizer. The polarization component has pointers allocated.

⟨*Particles: procedures*⟩+≡
```
    elemental subroutine particle_final (prt)
      type(particle_t), intent(inout) :: prt
      call polarization_final (prt%pol)
    end subroutine particle_final
```

I/O

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_write (prt, unit)
      type(particle_t), intent(in) :: prt
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      select case (prt%status)
      case (PRT_UNDEFINED);     write (u, "(1x, A)", advance="no")  "[-]"
      case (PRT_BEAM);          write (u, "(1x, A)", advance="no")  "[b]"
```

```
      case (PRT_INCOMING);    write (u, "(1x, A)", advance="no")  "[i]"
      case (PRT_OUTGOING);    write (u, "(1x, A)", advance="no")  "[o]"
      case (PRT_VIRTUAL);     write (u, "(1x, A)", advance="no")  "[v]"
      case (PRT_RESONANT);    write (u, "(1x, A)", advance="no")  "[r]"
      case (PRT_BEAM_REMNANT); write (u, "(1x, A)", advance="no")  "[x]"
      end select
      write (u, "(1x)", advance="no")
      call flavor_write (prt%flv, unit)
      call color_write (prt%col, unit)
      select case (prt%polarization)
      case (PRT_DEFINITE_HELICITY)
         call helicity_write (prt%hel, unit)
         write (u, *)
      case (PRT_GENERIC_POLARIZATION)
         write (u, *)
         call polarization_write (prt%pol, unit)
      case default
         write (u, *)
      end select
      write (u, *) "Momentum:"
      call vector4_write (prt%p, unit)
      write (u, "(1x,A)", advance="no")  "T = "
      write (u, *)  prt%p2
      if (allocated (prt%parent)) then
         if (size (prt%parent) /= 0) then
            write (u, "(1x,A,40(1x,I0))")  "Parents: ", prt%parent
         end if
      end if
      if (allocated (prt%child)) then
         if (size (prt%child) /= 0) then
            write (u, "(1x,A,40(1x,I0))")  "Children:", prt%child
         end if
      end if
    end subroutine particle_write
```

Binary I/O:

⟨*Particles: procedures*⟩+≡

```
    subroutine particle_write_raw (prt, u)
      type(particle_t), intent(in) :: prt
      integer, intent(in) :: u
      write (u) prt%status, prt%polarization
      call flavor_write_raw (prt%flv, u)
      call color_write_raw (prt%col, u)
      select case (prt%polarization)
      case (PRT_DEFINITE_HELICITY)
         call helicity_write_raw (prt%hel, u)
      case (PRT_GENERIC_POLARIZATION)
         call polarization_write_raw (prt%pol, u)
      end select
      call vector4_write_raw (prt%p, u)
      write (u) prt%p2
      write (u) allocated (prt%parent)
      if (allocated (prt%parent)) then
```

```
        write (u) size (prt%parent)
        write (u) prt%parent
     end if
     write (u) allocated (prt%child)
     if (allocated (prt%child)) then
        write (u) size (prt%child)
        write (u) prt%child
     end if
  end subroutine particle_write_raw

  subroutine particle_read_raw (prt, u, iostat)
    type(particle_t), intent(out) :: prt
    integer, intent(in) :: u
    integer, intent(out), optional :: iostat
    logical :: allocated_parent, allocated_child
    integer :: size_parent, size_child
    read (u, iostat=iostat) prt%status, prt%polarization
    call flavor_read_raw (prt%flv, u, iostat=iostat)
    call color_read_raw (prt%col, u, iostat=iostat)
    select case (prt%polarization)
    case (PRT_DEFINITE_HELICITY)
        call helicity_read_raw (prt%hel, u, iostat=iostat)
    case (PRT_GENERIC_POLARIZATION)
        call polarization_read_raw (prt%pol, u, iostat=iostat)
    end select
    call vector4_read_raw (prt%p, u, iostat=iostat)
    read (u, iostat=iostat) prt%p2
    read (u, iostat=iostat) allocated_parent
    if (allocated_parent) then
        read (u, iostat=iostat) size_parent
        allocate (prt%parent (size_parent))
        read (u, iostat=iostat) prt%parent
    end if
    read (u, iostat=iostat) allocated_child
    if (allocated_child) then
        read (u, iostat=iostat) size_child
        allocate (prt%child (size_child))
        read (u, iostat=iostat) prt%child
    end if
  end subroutine particle_read_raw
```

Transform a particle into a `hepmc_particle` object, including color and polarization. The HepMC status is equivalent to the HEPEVT status, in particular: 0 = null entry, 1 = physical particle, 2 = decayed/fragmented particle, 3 = other unphysical particle entry.

⟨*Particles: procedures*⟩+≡

```
  subroutine particle_to_hepmc (prt, hprt)
    type(particle_t), intent(in) :: prt
    type(hepmc_particle_t), intent(out) :: hprt
    integer :: hepmc_status
    select case (prt%status)
    case (PRT_UNDEFINED)
        hepmc_status = 0
```

```
      case (PRT_OUTGOING)
         hepmc_status = 1
      case (PRT_RESONANT)
         hepmc_status = 2
      case default
         hepmc_status = 3
      end select
      call hepmc_particle_init &
           (hprt, prt%p, flavor_get_pdg (prt%flv), hepmc_status)
      call hepmc_particle_set_color (hprt, prt%col)
      select case (prt%polarization)
      case (PRT_DEFINITE_HELICITY)
         call hepmc_particle_set_polarization (hprt, prt%hel)
      case (PRT_GENERIC_POLARIZATION)
         call hepmc_particle_set_polarization (hprt, prt%pol)
      end select
    end subroutine particle_to_hepmc
```

Setting contents

Reset the status code.

⟨*Particles: procedures*⟩+≡
```
    elemental subroutine particle_reset_status (prt, status)
      type(particle_t), intent(inout) :: prt
      integer, intent(in) :: status
      prt%status = status
    end subroutine particle_reset_status
```

The momentum is set independent of the quantum numbers.

⟨*Particles: procedures*⟩+≡
```
    elemental subroutine particle_set_momentum (prt, p)
      type(particle_t), intent(inout) :: prt
      type(vector4_t), intent(in) :: p
      prt%p = p
      prt%p2 = p ** 2
    end subroutine particle_set_momentum
```

Set resonance information. This should be done after momentum assignment, because we need to know wheter the particle is spacelike or timelike. The resonance flag is defined only for virtual particles.

⟨*Particles: procedures*⟩+≡
```
    elemental subroutine particle_set_resonance_flag (prt, resonant)
      type(particle_t), intent(inout) :: prt
      logical, intent(in) :: resonant
      select case (prt%status)
      case (PRT_VIRTUAL)
         if (resonant)  prt%status = PRT_RESONANT
      end select
    end subroutine particle_set_resonance_flag
```

Set children and parents information.

⟨Particles: procedures⟩+≡

```
subroutine particle_set_children (prt, idx)
  type(particle_t), intent(inout) :: prt
  integer, dimension(:), intent(in) :: idx
  if (allocated (prt%child))  deallocate (prt%child)
  allocate (prt%child (count (idx /= 0)))
  prt%child = pack (idx, idx /= 0)
end subroutine particle_set_children

subroutine particle_set_parents (prt, idx)
  type(particle_t), intent(inout) :: prt
  integer, dimension(:), intent(in) :: idx
  if (allocated (prt%parent))  deallocate (prt%parent)
  allocate (prt%parent (count (idx /= 0)))
  prt%parent = pack (idx, idx /= 0)
end subroutine particle_set_parents
```

Accessing contents

The status code.

⟨Particles: procedures⟩+≡

```
elemental function particle_get_status (prt) result (status)
  integer :: status
  type(particle_t), intent(in) :: prt
  status = prt%status
end function particle_get_status
```

Return true if the status is either `INCOMING`, `OUTGOING` or `RESONANT`. `BEAM` is kept, if `keep_beams` is set true.

⟨Particles: procedures⟩+≡

```
elemental function particle_is_real (prt, keep_beams) result (flag)
  logical :: flag, kb
  type(particle_t), intent(in) :: prt
  logical, intent(in), optional :: keep_beams
  kb = .false.
  if (present (keep_beams)) kb = keep_beams
  select case (prt%status)
  case (PRT_INCOMING, PRT_OUTGOING, PRT_RESONANT)
     flag = .true.
  case (PRT_BEAM)
     flag = kb
  case default
     flag = .false.
  end select
end function particle_is_real
```

Polarization status.

⟨Particles: procedures⟩+≡

```
elemental function particle_get_polarization_status (prt) result (status)
  integer :: status
```

```
      type(particle_t), intent(in) :: prt
      status = prt%polarization
    end function particle_get_polarization_status
```

Return the PDG code from the flavor component directly.

⟨*Particles: procedures*⟩+≡
```
    elemental function particle_get_pdg (prt) result (pdg)
      integer :: pdg
      type(particle_t), intent(in) :: prt
      pdg = flavor_get_pdg (prt%flv)
    end function particle_get_pdg
```

Return the color and anticolor quantum numbers.

⟨*Particles: procedures*⟩+≡
```
    function particle_get_color (prt) result (col)
      integer, dimension(2) :: col
      type(particle_t), intent(in) :: prt
      col(1) = color_get_col (prt%col)
      col(2) = color_get_acl (prt%col)
    end function particle_get_color
```

Return the polarization density matrix (as a shallow copy).

⟨*Particles: procedures*⟩+≡
```
    function particle_get_polarization (prt) result (pol)
      type(polarization_t) :: pol
      type(particle_t), intent(in) :: prt
      pol = prt%pol
    end function particle_get_polarization
```

Return the helicity (if defined and diagonal).

⟨*Particles: procedures*⟩+≡
```
    function particle_get_helicity (prt) result (hel)
      integer :: hel
      integer, dimension(2) :: hel_arr
      type(particle_t), intent(in) :: prt
      hel = 0
      if (helicity_is_defined (prt%hel) .and. &
          helicity_is_diagonal (prt%hel)) then
          hel_arr = helicity_get (prt%hel)
          hel = hel_arr (1)
      end if
    end function particle_get_helicity
```

Return the number of children/parents

⟨*Particles: procedures*⟩+≡
```
    function particle_get_n_parents (prt) result (n)
      integer :: n
      type(particle_t), intent(in) :: prt
      if (allocated (prt%parent)) then
         n = size (prt%parent)
      else
```

```
      n = 0
    end if
  end function particle_get_n_parents

  function particle_get_n_children (prt) result (n)
    integer :: n
    type(particle_t), intent(in) :: prt
    if (allocated (prt%child)) then
       n = size (prt%child)
    else
       n = 0
    end if
  end function particle_get_n_children
```

Return the array of parents/children.

⟨*Particles: procedures*⟩+≡

```
  function particle_get_parents (prt) result (parent)
    type(particle_t), intent(in) :: prt
    integer, dimension(:), allocatable :: parent
    if (allocated (prt%parent)) then
       allocate (parent (size (prt%parent)))
       parent = prt%parent
    else
       allocate (parent (0))
    end if
  end function particle_get_parents

  function particle_get_children (prt) result (child)
    type(particle_t), intent(in) :: prt
    integer, dimension(:), allocatable :: child
    if (allocated (prt%child)) then
       allocate (child (size (prt%child)))
       child = prt%child
    else
       allocate (child (0))
    end if
  end function particle_get_children
```

Return momentum and momentum squared.

⟨*Particles: procedures*⟩+≡

```
  function particle_get_momentum (prt) result (p)
    type(vector4_t) :: p
    type(particle_t), intent(in) :: prt
    p = prt%p
  end function particle_get_momentum

  function particle_get_p2 (prt) result (p2)
    real(default) :: p2
    type(particle_t), intent(in) :: prt
    p2 = prt%p2
  end function particle_get_p2
```

8.4.2 Particle sets

A particle set is what is usually called an event: an array of particles. The individual particle entries carry momentum, quantum numbers, polarization, and optionally connections. There is (also optionally) a correlated state-density matrix that maintains spin correlations that are lost in the individual particle entries.

⟨*Particles: public*⟩≡
```
public :: particle_set_t
```

⟨*Particles: types*⟩+≡
```
type :: particle_set_t
  private
  integer :: n_in  = 0
  integer :: n_vir = 0
  integer :: n_out = 0
  integer :: n_tot = 0
  type(particle_t), dimension(:), allocatable :: prt
  type(state_matrix_t) :: correlated_state
end type particle_set_t
```

A particle set can be initialized from an interaction or from a HepMC event record.

⟨*Particles: public*⟩+≡
```
public :: particle_set_init
```

⟨*Particles: interfaces*⟩+≡
```
interface particle_set_init
  module procedure particle_set_init_interaction
  module procedure particle_set_init_hepmc
end interface
```

When a particle set is initialized from a given interaction, we have to determine the branch within the original state matrix that fixes the particle quantum numbers. This is done with the appropriate probabilities, based on a random number x. The `mode` determines whether the individual particles become unpolarized, or take a definite (diagonal) helicity, or acquire single-particle polarization matrices. The flag `keep_correlations` tells whether the spin-correlation matrix is to be calculated and stored in addition to the particles. The flag `keep_virtual` tells whether virtual particles should be dropped. Note that if virtual particles are dropped, the spin-correlation matrix makes no sense, and parent-child relations are not set.

For a correct disentangling of color and flavor (in the presence of helicity), we consider two interactions. `int` has no color information, and is used to select a flavor state. Consequently, we trace over helicities here. `int_flows` contains color-flow and potentially helicity information, but is useful only after the flavor combination has been chosen. So this interaction is used to select helicity and color, but restricted to the selected flavor combination.

`int` and `int_flows` may be identical if there is only a single (or no) color flow. If there is just a single flavor combination, `x(1)` can be set to zero.

The current algorithm of evaluator convolution requires that the beam particles are assumed outgoing (in the beam interaction) and become virtual in all

derived interactions. In the particle set they should be re-identified as incoming.
The optional integer `n_incoming` can be used to perform this correction.

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_init_interaction &
     (particle_set, int, int_flows, mode, x, &
      keep_correlations, keep_virtual, n_incoming)
  type(particle_set_t), intent(out) :: particle_set
  type(interaction_t), intent(in), target :: int, int_flows
  integer, intent(in) :: mode
  real(default), dimension(2), intent(in) :: x
  logical, intent(in) :: keep_correlations, keep_virtual
  integer, intent(in), optional :: n_incoming
  type(state_matrix_t), dimension(:), allocatable, target :: flavor_state
  type(state_matrix_t), dimension(:), allocatable, target :: single_state
  integer :: n_in, n_vir, n_out, n_tot
  type(quantum_numbers_t), dimension(:,:), allocatable :: qn
  integer :: i, j
  if (present (n_incoming)) then
     n_in  = n_incoming
     n_vir = interaction_get_n_vir (int) - n_incoming
  else
     n_in  = interaction_get_n_in  (int)
     n_vir = interaction_get_n_vir (int)
  end if
  n_out = interaction_get_n_out (int)
  n_tot = interaction_get_n_tot (int)
  particle_set%n_in  = n_in
  particle_set%n_out = n_out
  if (keep_virtual) then
     particle_set%n_vir = n_vir
     particle_set%n_tot = n_tot
  else
     particle_set%n_vir = 0
     particle_set%n_tot = n_in + n_out
  end if
  call interaction_factorize (int, FM_IGNORE_HELICITY, x(1), flavor_state)
  allocate (qn (n_tot,1))
  do i = 1, n_tot
     qn(i,:) = state_matrix_get_quantum_numbers (flavor_state(i), 1)
  end do
  if (keep_correlations .and. keep_virtual) then
     call interaction_factorize (int_flows, &
          mode, x(2), single_state, particle_set%correlated_state, qn(:,1))
  else
     call interaction_factorize (int_flows, &
          mode, x(2), single_state, qn_in=qn(:,1))
  end if
  allocate (particle_set%prt (particle_set%n_tot))
  j = 1
  do i = 1, n_tot
     if (i <= n_in) then
        call particle_init &
             (particle_set%prt(j), single_state(i), PRT_INCOMING, mode)
     else if (i <= n_in + n_vir) then
```

```
          if (.not. keep_virtual)  cycle
          call particle_init &
               (particle_set%prt(j), single_state(i), PRT_VIRTUAL, mode)
       else
          call particle_init &
               (particle_set%prt(j), single_state(i), PRT_OUTGOING, mode)
       end if
       call particle_set_momentum &
            (particle_set%prt(j), interaction_get_momentum (int, i))
       if (keep_virtual) then
          call particle_set_children &
               (particle_set%prt(j), interaction_get_children (int, i))
          call particle_set_parents &
               (particle_set%prt(j), interaction_get_parents (int, i))
       end if
       j = j + 1
    end do
    if (keep_virtual) then
       call particle_set_resonance_flag &
            (particle_set%prt, interaction_get_resonance_flags (int))
    end if
    call state_matrix_final (flavor_state)
    call state_matrix_final (single_state)
  end subroutine particle_set_init_interaction
```

If a particle set is initialized from a HepMC event record, we have to specify
the treatment of polarization (unpolarized or density matrix) which is common
to all particles. Correlated polarization information is not available.

⟨*Particles: procedures*⟩+≡

```
  subroutine particle_set_init_hepmc (particle_set, evt, model, polarization)
    type(particle_set_t), intent(out) :: particle_set
    type(hepmc_event_t), intent(in) :: evt
    type(model_t), intent(in), target :: model
    integer, intent(in) :: polarization
    type(hepmc_event_particle_iterator_t) :: it
    type(hepmc_particle_t) :: prt
    integer, dimension(:), allocatable :: barcode
    integer :: n_tot, i
    n_tot = 0
    call hepmc_event_particle_iterator_init (it, evt)
    do while (hepmc_event_particle_iterator_is_valid (it))
       n_tot = n_tot + 1
       call hepmc_event_particle_iterator_advance (it)
    end do
    allocate (barcode (n_tot))
    call hepmc_event_particle_iterator_reset (it)
    do i = 1, n_tot
       barcode(i) = hepmc_particle_get_barcode &
            (hepmc_event_particle_iterator_get (it))
       call hepmc_event_particle_iterator_advance (it)
    end do
    allocate (particle_set%prt (n_tot))
    call hepmc_event_particle_iterator_reset (it)
```

```
      do i = 1, n_tot
         prt = hepmc_event_particle_iterator_get (it)
         call particle_init (particle_set%prt(i), &
              prt, model, polarization, barcode)
         call hepmc_event_particle_iterator_advance (it)
      end do
      call hepmc_event_particle_iterator_final (it)
      particle_set%n_tot = n_tot
   end subroutine particle_set_init_hepmc
```

Pointer components are hidden inside the particle polarization, and in the correlated state matrix.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_final
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_final (particle_set)
      type(particle_set_t), intent(inout) :: particle_set
      if (allocated (particle_set%prt))  call particle_final (particle_set%prt)
      call state_matrix_final (particle_set%correlated_state)
    end subroutine particle_set_final
```

Output (default format)

⟨*Particles: public*⟩+≡
```
    public :: particle_set_write
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_write (particle_set, unit)
      type(particle_set_t), intent(in) :: particle_set
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,A)") "Particle set:"
      if (particle_set%n_tot /= 0) then
         do i = 1, particle_set%n_tot
            write (u, "(1x,A,1x,I0)", advance="no") "Particle", i
            call particle_write (particle_set%prt(i), u)
         end do
         if (state_matrix_is_defined (particle_set%correlated_state)) then
            write (u, *) "Correlated state density matrix:"
            call state_matrix_write (particle_set%correlated_state, u)
         end if
      else
         write (u, "(3x,A)") "[empty]"
      end if
    end subroutine particle_set_write
```

8.4.3   I/O formats

Here, we define input/output of particle sets in various formats. This is the
right place since particle sets contain most of the event information.

All write/read routines take as first argument the object, as second argument the I/O unit which in this case is a mandatory argument. Then follow further event data.

Internal binary format

This format is supposed to contain the complete information, so the particle data set can be fully reconstructed. The exception is the model part of the particle flavors; this is unassigned for the flavor values read from file.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_write_raw
    public :: particle_set_read_raw
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_write_raw (particle_set, u)
      type(particle_set_t), intent(in) :: particle_set
      integer, intent(in) :: u
      integer :: i
      write (u) particle_set%n_in, particle_set%n_vir, particle_set%n_out
      write (u) particle_set%n_tot
      do i = 1, particle_set%n_tot
         call particle_write_raw (particle_set%prt(i), u)
      end do
      call state_matrix_write_raw (particle_set%correlated_state, u)
    end subroutine particle_set_write_raw

    subroutine particle_set_read_raw (particle_set, u, iostat)
      type(particle_set_t), intent(out) :: particle_set
      integer, intent(in) :: u
      integer, intent(out), optional :: iostat
      integer :: i
      read (u, iostat=iostat) &
           particle_set%n_in, particle_set%n_vir, particle_set%n_out
      read (u, iostat=iostat) particle_set%n_tot
      allocate (particle_set%prt (particle_set%n_tot))
      do i = 1, size (particle_set%prt)
         call particle_read_raw (particle_set%prt(i), u, iostat=iostat)
      end do
      call state_matrix_read_raw (particle_set%correlated_state, u, iostat=iostat)
    end subroutine particle_set_read_raw
```

Les Houches event format

This format consists of two common blocks and a file format. The actual interface is put in a separate module `event_formats`.

We first create a new particle set that contains only the particles that are supported by the LHEF format. These are: beam, incoming, resonant, outgoing. We drop particles with unknown, virtual or beam-remnant status.

From this set we fill the common block. Event information such as process ID and weight is not transferred here; this has to be done by the caller. The spin

information is set only if the particle has a unique mother, and if its polarization is fully defined.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_fill_hepeup
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_fill_hepeup (particle_set)
      type(particle_set_t), intent(in), target :: particle_set
      type(particle_t), pointer :: prt
      type(particle_set_t), target :: pset_reduced
      integer :: i, n_parents
      integer, dimension(1) :: i_mother
      call particle_set_reduce (particle_set, pset_reduced)
      call hepeup_init (pset_reduced%n_tot)
      do i = 1, pset_reduced%n_tot
         prt => pset_reduced%prt(i)
         call hepeup_set_particle (i, &
              particle_get_pdg (prt), &
              particle_get_status (prt), &
              particle_get_parents (prt), &
              particle_get_color (prt), &
              particle_get_momentum (prt), &
              particle_get_p2 (prt))
         n_parents = particle_get_n_parents (prt)
         if (n_parents == 1) then
            i_mother = particle_get_parents (prt)
            select case (particle_get_polarization_status (prt))
            case (PRT_GENERIC_POLARIZATION)
               call hepeup_set_particle_spin (i, &
                    particle_get_momentum (prt), &
                    particle_get_polarization (prt), &
                    particle_get_momentum (pset_reduced%prt(i_mother(1))))
            end select
         end if
      end do
      call particle_set_final (pset_reduced)
    end subroutine particle_set_fill_hepeup
```

The HEPEVT common block is quite similar, but does contain less information, e.g. no color flows (it was LEP time). The spin information is set only if the particle has a unique mother, and if its polarization is fully defined.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_fill_hepevt
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_fill_hepevt (particle_set, keep_beams)
      type(particle_set_t), intent(in), target :: particle_set
      type(particle_t), pointer :: prt
      type(particle_set_t), target :: pset_reduced
      logical, intent(in), optional :: keep_beams
      logical :: kb
      integer :: i, n_parents
      integer, dimension(1) :: i_mother
      kb = .false.
```

```
   if (present (keep_beams)) kb = keep_beams
   call particle_set_reduce (particle_set, pset_reduced, kb)
   call hepevt_init (pset_reduced%n_tot, pset_reduced%n_out)
   do i = 1, pset_reduced%n_tot
      prt => pset_reduced%prt(i)
      call hepevt_set_particle (i, &
            particle_get_pdg (prt), &
            particle_get_status (prt), &
            particle_get_parents (prt), &
            particle_get_children (prt), &
            particle_get_momentum (prt), &
            particle_get_p2 (prt), &
            particle_get_helicity (prt))
   end do
   call particle_set_final (pset_reduced)
end subroutine particle_set_fill_hepevt
```

HepMC format

The master output function fills a HepMC GenEvent object that is already initialized, but has no vertices in it.

We first set up the vertex lists and enter the vertices into the HepMC event. Then, we assign first all incoming particles and then all outgoing particles to their associated vertices. Particles which have neither parent nor children entries (this should not happen) are dropped.

⟨*Particles: public*⟩+≡

```
   public :: particle_set_fill_hepmc_event
```

⟨*Particles: procedures*⟩+≡

```
   subroutine particle_set_fill_hepmc_event (particle_set, evt)
     type(particle_set_t), intent(in) :: particle_set
     type(hepmc_event_t), intent(inout) :: evt
     type(hepmc_vertex_t), dimension(:), allocatable :: v
     type(hepmc_particle_t), dimension(:), allocatable :: hprt
     integer, dimension(:), allocatable :: v_from, v_to
     integer :: n_vertices, i
     allocate (v_from (particle_set%n_tot), v_to (particle_set%n_tot))
     call particle_set_assign_vertices (particle_set, v_from, v_to, n_vertices)
     allocate (v (n_vertices))
     do i = 1, n_vertices
        call hepmc_vertex_init (v(i))
        call hepmc_event_add_vertex (evt, v(i))
     end do
     allocate (hprt (particle_set%n_tot))
     do i = 1, particle_set%n_tot
        if (v_to(i) /= 0 .or. v_from(i) /= 0) then
           call particle_to_hepmc (particle_set%prt(i), hprt(i))
        end if
     end do
     do i = 1, particle_set%n_tot
        if (v_to(i) /= 0) then
           call hepmc_vertex_add_particle_in (v(v_to(i)), hprt(i))
        end if
```

746

```
         end do
      do i = 1, particle_set%n_tot
         if (v_from(i) /= 0) then
            call hepmc_vertex_add_particle_out (v(v_from(i)), hprt(i))
         end if
      end do
   end subroutine particle_set_fill_hepmc_event
```

Tools

Given a particle list, reset status codes.

⟨*Particles: public*⟩+≡
```
   public :: particle_set_reset_status
```

⟨*Particles: procedures*⟩+≡
```
   subroutine particle_set_reset_status (particle_set, index, status)
     type(particle_set_t), intent(inout) :: particle_set
     integer, dimension(:), intent(in) :: index
     integer, intent(in) :: status
     integer :: i
     if (allocated (particle_set%prt)) then
        do i = 1, size (index)
           call particle_reset_status (particle_set%prt(index(i)), status)
        end do
     end if
   end subroutine particle_set_reset_status
```

Find parents/children of a particular particle recursively; the search terminates
if a parent/child has status BEAM, INCOMING, OUTGOING or RESONANT.

⟨*Particles: procedures*⟩+≡
```
   function particle_set_get_real_parents (pset, i, keep_beams) result (parent)
     integer, dimension(:), allocatable :: parent
     type(particle_set_t), intent(in) :: pset
     integer, intent(in) :: i
     logical, intent(in), optional :: keep_beams
     logical, dimension(:), allocatable :: is_real
     logical, dimension(:), allocatable :: is_parent, is_real_parent
     logical :: kb
     integer :: j, k
     kb = .false.
     if (present (keep_beams)) kb = keep_beams
     allocate (is_real (pset%n_tot))
     is_real = particle_is_real (pset%prt(i), kb)
     allocate (is_parent (pset%n_tot), is_real_parent (pset%n_tot))
     is_real_parent = .false.
     is_parent = .false.
     is_parent(particle_get_parents(pset%prt(i))) = .true.
     do while (any (is_parent))
        where (is_real .and. is_parent)
           is_real_parent = .true.
           is_parent = .false.
        end where
```

```fortran
      do j = size (is_parent), 1, -1
         if (is_parent(j)) then
            is_parent(particle_get_parents(pset%prt(j))) = .true.
         end if
      end do
   end do
   allocate (parent (count (is_real_parent)))
   j = 0
   do k = 1, size (is_parent)
      if (is_real_parent(k)) then
         j = j + 1
         parent(j) = k
      end if
   end do
end function particle_set_get_real_parents

function particle_set_get_real_children (pset, i, keep_beams) result (child)
   integer, dimension(:), allocatable :: child
   type(particle_set_t), intent(in) :: pset
   integer, intent(in) :: i
   logical, dimension(:), allocatable :: is_real
   logical, dimension(:), allocatable :: is_child, is_real_child
   logical, intent(in), optional :: keep_beams
   integer :: j, k
   logical :: kb
   kb = .false.
   if (present (keep_beams)) kb = keep_beams
   allocate (is_real (pset%n_tot))
   is_real = particle_is_real (pset%prt(i), kb)
   allocate (is_child (pset%n_tot), is_real_child (pset%n_tot))
   is_real_child = .false.
   is_child = .false.
   is_child(particle_get_children(pset%prt(i))) = .true.
   do while (any (is_child))
      where (is_real .and. is_child)
         is_real_child = .true.
         is_child = .false.
      end where
      do j = 1, size (is_child)
         if (is_child(j)) then
            is_child(particle_get_children(pset%prt(j))) = .true.
         end if
      end do
   end do
   allocate (child (count (is_real_child)))
   j = 0
   do k = 1, size (is_child)
      if (is_real_child(k)) then
         j = j + 1
         child(j) = k
      end if
   end do
end function particle_set_get_real_children
```

Get the `n_tot`, `n_in`, and `n_out` values out of the particle set.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_get_n_out, particle_set_get_n_in, &
                particle_set_get_n_tot
```

⟨*Particles: procedures*⟩+≡
```
    function particle_set_get_n_out (pset) result (n_out)
       type(particle_set_t), intent(in) :: pset
       integer :: n_out
       n_out = pset%n_out
    end function particle_set_get_n_out

    function particle_set_get_n_in (pset) result (n_in)
       type(particle_set_t), intent(in) :: pset
       integer :: n_in
       n_in = pset%n_in
    end function particle_set_get_n_in

    function particle_set_get_n_tot (pset) result (n_tot)
       type(particle_set_t), intent(in) :: pset
       integer :: n_tot
       n_tot = pset%n_tot
    end function particle_set_get_n_tot
```

Reduce a particle set to the essential entries. The entries kept are those with status `INCOMING`, `OUTGOING` or `RESONANT`. `BEAM` is kept if `keep_beams` is true. Other entries are skipped. The correlated state matrix, if any, is also ignored.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_reduce
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_reduce (pset_in, pset_out, keep_beams)
      type(particle_set_t), intent(in) :: pset_in
      type(particle_set_t), intent(out) :: pset_out
      logical, intent(in), optional :: keep_beams
      integer, dimension(:), allocatable :: status, map
      integer :: i, j
      logical :: kb
      kb = .false.
      if (present (keep_beams)) kb = keep_beams
      allocate (status (pset_in%n_tot))
      status = particle_get_status (pset_in%prt)
      if (kb) then
         pset_out%n_in  = count (status == PRT_BEAM)
         pset_out%n_vir = count (status == PRT_INCOMING) &
               + count (status == PRT_RESONANT)
      else
         pset_out%n_in  = count (status == PRT_INCOMING)
         pset_out%n_vir = count (status == PRT_RESONANT)
      end if
      pset_out%n_out = count (status == PRT_OUTGOING)
      pset_out%n_tot = pset_out%n_in + pset_out%n_vir + pset_out%n_out
      allocate (pset_out%prt (pset_out%n_tot))
      allocate (map (pset_in%n_tot))
```

```
      map = 0
      j = 0
      if (kb) call copy_particles (PRT_BEAM)
      call copy_particles (PRT_INCOMING)
      call copy_particles (PRT_RESONANT)
      call copy_particles (PRT_OUTGOING)
      do i = 1, pset_in%n_tot
         if (map(i) == 0)  cycle
! triggers nagfor bug!
!         call particle_set_parents (pset_out%prt(map(i)), &
!              map (particle_set_get_real_parents (pset_in, i)))
!         call particle_set_children (pset_out%prt(map(i)), &
!              map (particle_set_get_real_children (pset_in, i)))
! workaround:
         call particle_set_parents (pset_out%prt(map(i)), &
              particle_set_get_real_parents (pset_in, i, kb))
         call particle_set_parents (pset_out%prt(map(i)), &
              map (pset_out%prt(map(i))%parent))
         call particle_set_children (pset_out%prt(map(i)), &
              particle_set_get_real_children (pset_in, i, kb))
         call particle_set_children (pset_out%prt(map(i)), &
              map (pset_out%prt(map(i))%child))
      end do
   contains
     subroutine copy_particles (stat)
       integer, intent(in) :: stat
       integer :: i
       do i = 1, pset_in%n_tot
          if (status(i) == stat) then
             j = j + 1
             map(i) = j
             call particle_init (pset_out%prt(j), pset_in%prt(i))
          end if
       end do
     end subroutine copy_particles
   end subroutine particle_set_reduce
```

This procedure reconstructs an array of vertex indices from the parent-child information in the particle entries, according to the HepMC scheme. For each particle, we determine which vertex it comes from and which vertex it goes to. We return the two arrays and the maximum vertex index.

For each particle in the list, we first check its parents. If for any parent the vertex where it goes to is already known, this vertex index is assigned as the current 'from' vertex. Otherwise, a new index is created, assigned as the current 'from' vertex, and as the 'to' vertex for all parents.

Then, the analogous procedure is done for the children.

⟨*Particles: procedures*⟩+≡
```
   subroutine particle_set_assign_vertices &
       (particle_set, v_from, v_to, n_vertices)
     type(particle_set_t), intent(in) :: particle_set
     integer, dimension(:), intent(out) :: v_from, v_to
     integer, intent(out) :: n_vertices
     integer, dimension(:), allocatable :: parent, child
```

```
      integer :: n_parents, n_children, vf, vt
      integer :: i, j, v
      v_from = 0
      v_to = 0
      vf = 0
      vt = 0
      do i = 1, particle_set%n_tot
         n_parents = particle_get_n_parents (particle_set%prt(i))
         if (n_parents /= 0) then
            allocate (parent (n_parents))
            parent = particle_get_parents (particle_set%prt(i))
            SCAN_PARENTS: do j = 1, size (parent)
               v = v_to(parent(j))
               if (v /= 0) then
                  v_from(i) = v;  exit SCAN_PARENTS
               end if
            end do SCAN_PARENTS
            if (v_from(i) == 0) then
               vf = vf + 1;  v_from(i) = vf
               v_to(parent) = vf
            end if
            deallocate (parent)
         end if
         n_children = particle_get_n_children (particle_set%prt(i))
         if (n_children /= 0) then
            allocate (child (n_children))
            child = particle_get_children (particle_set%prt(i))
            SCAN_CHILDREN: do j = 1, size (child)
               v = v_from(child(j))
               if (v /= 0) then
                  v_to(i) = v;  exit SCAN_CHILDREN
               end if
            end do SCAN_CHILDREN
            if (v_to(i) == 0) then
               vt = vt + 1;  v_to(i) = vt
               v_from(child) = vt
            end if
            deallocate (child)
         end if
      end do
      n_vertices = max (vf, vt)
   end subroutine particle_set_assign_vertices
```

8.4.4 Expression interface

This converts a `particle_set` object as defined here to a more concise `prt_list` object that can be used as the event root of an expression. In particular, the latter lacks virtual particles, spin correlations and parent-child relations.

TODO: The `prt_list` also lacks helicity/pol info.

⟨*Particles: public*⟩+≡
```
   public :: particle_set_to_prt_list
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_to_prt_list (particle_set, prt_list)
      type(particle_set_t), intent(in), target :: particle_set
      type(prt_list_t), intent(out) :: prt_list
      type(particle_t), pointer :: prt
      integer :: i, k
      call prt_list_init (prt_list)
      call prt_list_reset (prt_list, particle_set%n_in + particle_set%n_out)
      k = 0
      do i = 1, particle_set%n_tot
         prt => particle_set%prt(i)
         select case (particle_get_status (prt))
         case (PRT_INCOMING)
            k = k + 1
            call prt_list_set_incoming (prt_list, k, &
                 particle_get_pdg (prt), &
                 particle_get_momentum (prt), &
                 particle_get_p2 (prt))
         case (PRT_OUTGOING)
            k = k + 1
            call prt_list_set_outgoing (prt_list, k, &
                 particle_get_pdg (prt), &
                 particle_get_momentum (prt), &
                 particle_get_p2 (prt))
         end select
      end do
    end subroutine particle_set_to_prt_list
```

8.4.5   Test

Set up a chain of production and decay and factorize the result into particles.
The process is $d\bar{d} \to Z \to q\bar{q}$.

⟨*Particles: public*⟩+≡
```
    public :: particles_test
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particles_test
      use os_interface, only: os_data_t
      type(os_data_t) :: os_data
      type(model_t), pointer :: model
      type(flavor_t), dimension(3) :: flv
      type(color_t), dimension(3) :: col
      type(helicity_t), dimension(3) :: hel
      type(quantum_numbers_t), dimension(3) :: qn
      type(vector4_t), dimension(3) :: p
      type(interaction_t), target :: int1, int2
      type(quantum_numbers_mask_t) :: qn_mask_conn, qn_rest
      type(evaluator_t), target :: eval
      type(interaction_t), pointer :: int
      type(particle_set_t) :: particle_set1, particle_set2
      type(particle_set_t) :: particle_set3, particle_set4
      type(hepmc_event_t) :: hepmc_event
      type(hepmc_iostream_t) :: iostream
```

```
type(prt_list_t) :: prt_list
integer :: u
print *, "*** Read model file"
call syntax_model_file_init ()
call model_list_read_model &
     (var_str("QCD"), var_str("test.mdl"), os_data, model)
print *
print *, "*** Setup production process ***"
call interaction_init (int1, 2, 0, 1, set_relations=.true.)
call flavor_init (flv, (/1, -1, 23/), model)
call helicity_init (hel(3), 1, 1)
call quantum_numbers_init (qn, flv, hel)
call interaction_add_state (int1, qn, value=(0.25_default, 0._default))
call helicity_init (hel(3), 1,-1)
call quantum_numbers_init (qn, flv, hel)
call interaction_add_state (int1, qn, value=(0._default, 0.25_default))
call helicity_init (hel(3),-1, 1)
call quantum_numbers_init (qn, flv, hel)
call interaction_add_state (int1, qn, value=(0._default,-0.25_default))
call helicity_init (hel(3),-1,-1)
call quantum_numbers_init (qn, flv, hel)
call interaction_add_state (int1, qn, value=(0.25_default, 0._default))
call helicity_init (hel(3), 0, 0)
call quantum_numbers_init (qn, flv, hel)
call interaction_add_state (int1, qn, value=(0.5_default, 0._default))
call interaction_freeze (int1)
p(1) = vector4_moving (45._default, 45._default, 3)
p(2) = vector4_moving (45._default,-45._default, 3)
p(3) = p(1) + p(2)
call interaction_set_momenta (int1, p)
print *
print *, "*** Setup decay process ***"
call interaction_init (int2, 1, 0, 2, set_relations=.true.)
call flavor_init (flv, (/23, 1, -1/), model)
call color_init_col_acl (col, (/ 0, 501, 0 /), (/ 0, 0, 501 /))
call helicity_init (hel, (/ 1, 1, 1/), (/ 1, 1, 1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(1._default, 0._default))
call helicity_init (hel, (/ 1, 1, 1/), (/-1,-1,-1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(0._default, 0.1_default))
call helicity_init (hel, (/-1,-1,-1/), (/ 1, 1, 1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(0._default,-0.1_default))
call helicity_init (hel, (/-1,-1,-1/), (/-1,-1,-1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(1._default, 0._default))
call helicity_init (hel, (/ 0, 1,-1/), (/ 0, 1,-1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(4._default, 0._default))
call helicity_init (hel, (/ 0,-1, 1/), (/ 0, 1,-1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(2._default, 0._default))
call helicity_init (hel, (/ 0, 1,-1/), (/ 0,-1, 1/))
```

```
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(2._default, 0._default))
call helicity_init (hel, (/ 0,-1, 1/), (/ 0,-1, 1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(4._default, 0._default))
call flavor_init (flv, (/23, 2, -2/), model)
call helicity_init (hel, (/ 0, 1,-1/), (/ 0, 1,-1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(0.5_default, 0._default))
call helicity_init (hel, (/ 0,-1, 1/), (/ 0,-1, 1/))
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(0.5_default, 0._default))
call interaction_freeze (int2)
p(2) = vector4_moving (45._default, 45._default, 2)
p(3) = vector4_moving (45._default,-45._default, 2)
call interaction_set_momenta (int2, p)
call interaction_set_source_link (int2, 1, int1, 3)
call interaction_write (int1)
call interaction_write (int2)
print *
print *, "*** Concatenate production and decay ***"
call evaluator_init_product (eval, int1, int2, qn_mask_conn, &
    connections_are_resonant=.true.)
call evaluator_receive_momenta (eval)
call evaluator_evaluate (eval)
call evaluator_write (eval)
print *
print *, "*** Factorize as particle list (complete, polarized) ***"
int => evaluator_get_int_ptr (eval)
call particle_set_init &
    (particle_set1, int, int, FM_FACTOR_HELICITY, &
     (/0.2_default, 0.2_default/), .false., .true.)
call particle_set_write (particle_set1)
print *
print *, "*** Write this to HEPEUP and print as LHEF format ***"
call les_houches_events_write_header ()
call heprup_init ((/2212, 2212/), (/7.e3_default, 7.e3_default/), &
    n_processes=1, unweighted=.true., negative_weights=.false.)
call heprup_set_process_parameters (1, 1)
call heprup_write_lhef ()
call particle_set_fill_hepeup (particle_set1)
call hepeup_write_lhef ()
call les_houches_events_write_footer ()
print *
print *, "*** Factorize as particle list (in/out only, selected helicity) ***"
int => evaluator_get_int_ptr (eval)
call particle_set_init &
    (particle_set2, int, int, FM_SELECT_HELICITY, &
     (/0.9_default, 0.9_default/), .false., .false.)
call particle_set_write (particle_set2)
call particle_set_final (particle_set2)
print *
print *, "*** Factorize as particle list (complete, selected helicity) ***"
int => evaluator_get_int_ptr (eval)
```

```
call particle_set_init &
     (particle_set2, int, int, FM_SELECT_HELICITY, &
      (/0.7_default, 0.7_default/), .false., .true.)
call particle_set_write (particle_set2)
print *
print *, "*** Write to HepMC, print, and output to particles_test.hepmc.dat ***"
call hepmc_event_init (hepmc_event, 11, 127)
call particle_set_fill_hepmc_event (particle_set2, hepmc_event)
call hepmc_event_print (hepmc_event)
call hepmc_iostream_open_out &
     (iostream , var_str ("particles_test.hepmc.dat"))
call hepmc_iostream_write_event (iostream, hepmc_event)
call hepmc_iostream_close (iostream)
print *
print *, "*** Recover from HepMC file ***"
call particle_set_final (particle_set2)
call hepmc_event_final (hepmc_event)
call hepmc_event_init (hepmc_event)
call hepmc_iostream_open_in &
     (iostream , var_str ("particles_test.hepmc.dat"))
call hepmc_iostream_read_event (iostream, hepmc_event)
call hepmc_iostream_close (iostream)
call particle_set_init (particle_set2, &
     hepmc_event, model, PRT_DEFINITE_HELICITY)
call particle_set_write (particle_set2)
print *
print *, "*** Factorize (complete, polarized, correlated); write and read again ***"
int => evaluator_get_int_ptr (eval)
call particle_set_init &
     (particle_set3, int, int, FM_FACTOR_HELICITY, &
      (/0.7_default, 0.7_default/), .true., .true.)
call particle_set_write (particle_set3)
u = free_unit ()
open (u, action="readwrite", form="unformatted", status="scratch")
call particle_set_write_raw (particle_set3, u)
rewind (u)
call particle_set_read_raw (particle_set4, u)
close (u)
print *
call particle_set_write (particle_set4)
print *
print *, "*** Transform to a prt_list object ***"
call particle_set_to_prt_list (particle_set4, prt_list)
call prt_list_write (prt_list)
print *
print *, "*** Cleanup ***"
call particle_set_final (particle_set1)
call particle_set_final (particle_set2)
call particle_set_final (particle_set3)
call particle_set_final (particle_set4)
call evaluator_final (eval)
call interaction_final (int1)
call interaction_final (int2)
call hepmc_event_final (hepmc_event)
```

```
end subroutine particles_test
```

Chapter 9

Initial State

9.1 Beams for collisions and decays

⟨beams.f90⟩≡
  ⟨*File header*⟩

  module beams

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
    use constants !NODEP!
  ⟨*Use file utils*⟩
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use md5
    use models
    use flavors
    use colors
    use polarizations
    use quantum_numbers
    use state_matrices
    use interactions

  ⟨*Standard module head*⟩

  ⟨*Beams: public*⟩

  ⟨*Beams: types*⟩

  ⟨*Beams: interfaces*⟩

  contains

  ⟨*Beams: procedures*⟩

  end module beams

757

9.1.1 Beam data

The beam data type contains beam data for one or two beams, depending on whether we are dealing with beam collisions or particle decay. In addition, it holds the c.m. energy `sqrts`, the Lorentz transformation `L` that transforms the c.m. system into the lab system, and the pair of c.m. momenta.

⟨*Beams: public*⟩≡

```
public :: beam_data_t
```

⟨*Beams: types*⟩≡

```
type :: beam_data_t
   logical :: initialized = .false.
   integer :: n = 0
   type(flavor_t), dimension(:), allocatable :: flv
   real(default), dimension(:), allocatable :: mass
   type(polarization_t), dimension(:), allocatable :: pol
   logical :: lab_is_cm_frame = .true.
   type(vector4_t), dimension(:), allocatable :: p_cm
   type(vector4_t), dimension(:), allocatable :: p
   type(lorentz_transformation_t), pointer  :: L_cm_to_lab => null ()
   real(default) :: sqrts = 0
   character(32) :: md5sum = ""
end type beam_data_t
```

Generic initializer. This is called by the specific initializers below. Initialize either for decay or for collision.

⟨*Beams: procedures*⟩≡

```
subroutine beam_data_init (beam_data, n)
  type(beam_data_t), intent(out) :: beam_data
  integer, intent(in) :: n
  beam_data%n = n
  allocate (beam_data%flv (n))
  allocate (beam_data%mass (n))
  allocate (beam_data%pol (n))
  allocate (beam_data%p_cm (n))
  allocate (beam_data%p (n))
  beam_data%initialized = .true.
end subroutine beam_data_init
```

Finalizer: needed for the polarization components of the beams.

⟨*Beams: public*⟩+≡

```
public :: beam_data_final
```

⟨*Beams: procedures*⟩+≡

```
subroutine beam_data_final (beam_data)
  type(beam_data_t), intent(inout) :: beam_data
  beam_data%initialized = .false.
  if (allocated (beam_data%pol))  call polarization_final (beam_data%pol)
  if (associated (beam_data%L_cm_to_lab))  deallocate (beam_data%L_cm_to_lab)
end subroutine beam_data_final
```

The verbose (default) version is for debugging. The short version is for screen output in the UI.

⟨*Beams: public*⟩+≡
```
    public :: beam_data_write
```

⟨*Beams: procedures*⟩+≡
```
    subroutine beam_data_write (beam_data, unit, verbose, write_md5sum)
      type(beam_data_t), intent(in) :: beam_data
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose, write_md5sum
      integer :: prt_name_len
      logical :: verb, write_md5
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      verb = .true.;  if (present (verbose))  verb = verbose
      write_md5 = verb;  if (present (write_md5sum)) write_md5 = write_md5sum
      if (.not. beam_data%initialized) then
         write (u, "(A)") "Beam data: [undefined]"
         return
      end if
      prt_name_len = maxval (len (flavor_get_name (beam_data%flv)))
      select case (beam_data%n)
      case (1)
         write (u, "(A)") "Beam data (decay):"
         if (verb) then
            call write_prt (1)
            call polarization_write (beam_data%pol(1), u)
            write (u, *) "R.f. momentum:"
            call vector4_write (beam_data%p_cm(1), u)
            write (u, *) "Lab momentum:"
            call vector4_write (beam_data%p(1), u)
         else
            call write_prt (1)
         end if
      case (2)
         write (u, "(A)") "Beam data (collision):"
         if (verb) then
            call write_prt (1)
            call polarization_write (beam_data%pol(1), u)
            call write_prt (2)
            call polarization_write (beam_data%pol(2), u)
            call write_sqrts
            write (u, *) "C.m. momenta:"
            call vector4_write (beam_data%p_cm(1), u)
            call vector4_write (beam_data%p_cm(2), u)
            write (u, *) "Lab momenta:"
            call vector4_write (beam_data%p(1), u)
            call vector4_write (beam_data%p(2), u)
         else
            call write_prt (1)
            call write_prt (2)
            call write_sqrts
         end if
      end select
```

```
      if (associated (beam_data%L_cm_to_lab)) &
           call lorentz_transformation_write (beam_data%L_cm_to_lab, u)
      if (write_md5) then
         write (u, *) "MD5 sum: ", beam_data%md5sum
      end if
    contains
      subroutine write_sqrts
        character(80) :: sqrts_str
        write (sqrts_str, "(1PG21.15)")  beam_data%sqrts
        write (u, "(1x,A)")  "sqrts = " // trim (adjustl (sqrts_str)) // " GeV"
      end subroutine write_sqrts
      subroutine write_prt (i)
        integer, intent(in) :: i
        character(80) :: name_str, mass_str
        write (name_str, "(A)")  char (flavor_get_name (beam_data%flv(i)))
        write (mass_str, "(1PG15.8)")  beam_data%mass(i)
        write (u, "(1x,A)")  name_str(:prt_name_len) // "  (mass = " &
              // trim (adjustl (mass_str)) // " GeV)"
      end subroutine write_prt
    end subroutine beam_data_write
```

Return initialization status:

⟨*Beams: public*⟩+≡
```
    public :: beam_data_are_valid
```

⟨*Beams: procedures*⟩+≡
```
    function beam_data_are_valid (beam_data) result (flag)
      logical :: flag
      type(beam_data_t), intent(in) :: beam_data
      flag = beam_data%initialized
    end function beam_data_are_valid
```

Check whether beam data agree with the current values of relevant parameters.

⟨*Beams: public*⟩+≡
```
    public :: beam_data_check_scattering
```

⟨*Beams: procedures*⟩+≡
```
    subroutine beam_data_check_scattering (beam_data, sqrts)
      type(beam_data_t), intent(in) :: beam_data
      real(default), intent(in) :: sqrts
      if (beam_data_are_valid (beam_data)) then
         if (sqrts /= beam_data%sqrts) then
            call msg_error ("Current setting of sqrts is inconsistent " &
                 // "with beam setup (ignored).")
         end if
      else
         call msg_bug ("Beam setup: invalid beam data")
      end if
    end subroutine beam_data_check_scattering
```

Return the number of beams (1 for decays, 2 for collisions).

⟨*Beams: public*⟩+≡
```
    public :: beam_data_get_n_in
```

⟨Beams: procedures⟩+≡

```
function beam_data_get_n_in (beam_data) result (n_in)
  integer :: n_in
  type(beam_data_t), intent(in) :: beam_data
  n_in = beam_data%n
end function beam_data_get_n_in
```

Return the beam energies

⟨Beams: public⟩+≡

```
public :: beam_data_get_energy
```

⟨Beams: procedures⟩+≡

```
function beam_data_get_energy (beam_data) result (e)
  real(default), dimension(:), allocatable :: e
  type(beam_data_t), intent(in) :: beam_data
  allocate (e (beam_data%n))
  if (beam_data%initialized) then
     e = energy (beam_data%p)
  else
     e = 0
  end if
end function beam_data_get_energy
```

Return a MD5 checksum for beam data. If no checksum is present (because beams have not been initialized), compute the checksum of the sqrts value.

⟨Beams: public⟩+≡

```
public :: beam_data_get_md5sum
```

⟨Beams: procedures⟩+≡

```
function beam_data_get_md5sum (beam_data, sqrts) result (md5sum_beams)
  type(beam_data_t), intent(in) :: beam_data
  real(default), intent(in) :: sqrts
  character(32) :: md5sum_beams
  character(80) :: buffer
  if (beam_data%md5sum /= "") then
     md5sum_beams = beam_data%md5sum
  else
     write (buffer, *)  sqrts
     md5sum_beams = md5sum (buffer)
  end if
end function beam_data_get_md5sum
```

9.1.2 Initializers: collisions

This is the simplest one: just the two flavors, c.m. energy, polarization. Color is inferred from flavor. Beam momenta and c.m. momenta coincide.

⟨Beams: public⟩+≡

```
public :: beam_data_init_sqrts
```

⟨*Beams: procedures*⟩+≡

```
subroutine beam_data_init_sqrts &
    (beam_data, sqrts, flv, pol, p_cm, p_cm_theta, p_cm_phi)
  type(beam_data_t), intent(out) :: beam_data
  real(default), intent(in) :: sqrts
  type(flavor_t), dimension(:), intent(in) :: flv
  type(polarization_t), dimension(:), intent(in), optional :: pol
  real(default), intent(in), optional :: p_cm, p_cm_theta, p_cm_phi
  real(default), dimension(size(flv)) :: E, p
  integer :: i
  call beam_data_init (beam_data, size (flv))
  beam_data%sqrts = sqrts
  beam_data%lab_is_cm_frame = &
      .not. present (p_cm) .and. .not. present (p_cm_theta)
  select case (beam_data%n)
  case (1)
     if (present (p_cm)) then
        E = sqrt (sqrts**2 + p_cm**2)
        p = p_cm
     else
        E = sqrts;  p = 0
     end if
     beam_data%p_cm = vector4_moving (E, p, 3)
  case (2)
     beam_data%p_cm = colliding_momenta (sqrts, flavor_get_mass (flv), p_cm)
  end select
  if (present (p_cm_theta)) then
     beam_data%p_cm = rotation (p_cm_theta, 2) * beam_data%p_cm
     if (present (p_cm_phi)) then
        beam_data%p_cm = rotation (p_cm_phi, 3) * beam_data%p_cm
     end if
  end if
  do i = 1, beam_data%n
     beam_data%flv(i) = flv(i)
     beam_data%mass(i) = flavor_get_mass (flv(i))
     beam_data%p(i) = beam_data%p_cm(i)
     if (present (pol)) then
        beam_data%pol(i) = pol(i)
     else
        call polarization_init_unpolarized (beam_data%pol(i), flv(i))
     end if
  end do
  call beam_data_compute_md5sum (beam_data)
end subroutine beam_data_init_sqrts
```

9.1.3 Initializers: decays

This is the simplest one: decay in rest frame. We need just flavor and polarization. Color is inferred from flavor. Beam momentum and c.m. momentum coincide.

⟨*Beams: public*⟩+≡

```
public :: beam_data_init_decay
```

⟨*Beams: procedures*⟩+≡

```
    subroutine beam_data_init_decay &
         (beam_data, flv, pol, p_cm, p_cm_theta, p_cm_phi)
      type(beam_data_t), intent(out) :: beam_data
      type(flavor_t), dimension(1), intent(in) :: flv
      type(polarization_t), dimension(1), intent(in), optional :: pol
      real(default), intent(in), optional :: p_cm, p_cm_theta, p_cm_phi
      real(default), dimension(1) :: m
      type(polarization_t), dimension(1) :: polarization
      m = flavor_get_mass (flv)
      if (present (pol)) then
         call beam_data_init_sqrts &
              (beam_data, m(1), flv, pol, p_cm, p_cm_theta, p_cm_phi)
      else
         call polarization_init_trivial (polarization(1), flv(1))
         call beam_data_init_sqrts &
              (beam_data, m(1), flv, polarization, p_cm, p_cm_theta, p_cm_phi)
      end if
    end subroutine beam_data_init_decay
```

9.1.4 Compute MD5 sum

The MD5 sum is stored within the beam-data record, so it can be checked for integrity in subsequent runs.

⟨*Beams: procedures*⟩+≡

```
    subroutine beam_data_compute_md5sum (beam_data)
      type(beam_data_t), intent(inout) :: beam_data
      integer :: unit
      unit = free_unit ()
      open (unit = unit, status = "scratch", action = "readwrite")
      call beam_data_write (beam_data, unit, write_md5sum = .false.)
      rewind (unit)
      beam_data%md5sum = md5sum (unit)
      close (unit)
    end subroutine beam_data_compute_md5sum
```

9.1.5 Sanity check

After the beams have been set, the initial-particle masses may have been modified. This can be checked here.

⟨*Beams: public*⟩+≡

```
    public :: beam_data_masses_are_consistent
```

⟨*Beams: procedures*⟩+≡

```
    function beam_data_masses_are_consistent (beam_data) result (flag)
      logical :: flag
      type(beam_data_t), intent(in) :: beam_data
      flag = all (beam_data%mass == flavor_get_mass (beam_data%flv))
    end function beam_data_masses_are_consistent
```

9.1.6 The beams type

Beam objects are interaction objects that contain the actual beam data including polarization and density matrix. For collisions, the beam object actually contains two beams.

⟨*Beams: public*⟩+≡

```
public :: beam_t
```

⟨*Beams: types*⟩+≡

```
type :: beam_t
   private
   type(interaction_t) :: int
end type beam_t
```

The constructor contains code that converts beam data into the (entangled) particle-pair quantum state. First, we set the number of particles and polarization mask. (The polarization mask is handed over to all later interactions, so if helicity is diagonal or absent, this fact is used when constructing the hard-interaction events.) Then, we construct the entangled state that combines helicity, flavor and color of the two particles (where flavor and color are unique, while several helicity states are possible). Then, we transfer this state together with the associated values from the spin density matrix into the `interaction_t` object.

⟨*Beams: public*⟩+≡

```
public :: beam_init
```

⟨*Beams: procedures*⟩+≡

```
subroutine beam_init (beam, beam_data)
  type(beam_t), intent(out) :: beam
  type(beam_data_t), intent(in), target :: beam_data
  type(quantum_numbers_mask_t), dimension(beam_data%n) :: mask
  type(state_matrix_t), target :: state_hel, state_fc, state_tmp
  type(state_iterator_t) :: it_hel, it_tmp
  type(quantum_numbers_t), dimension(:), allocatable :: qn
  mask = new_quantum_numbers_mask (.false., .false., &
       .not. polarization_is_polarized (beam_data%pol), &
       mask_hd = polarization_is_diagonal (beam_data%pol))
  call interaction_init &
       (beam%int, 0, 0, beam_data%n, mask=mask, store_values=.true.)
  call combine_polarization_states (beam_data%pol, state_hel)
  allocate (qn (beam_data%n))
  call quantum_numbers_init &
       (qn, beam_data%flv, color_from_flavor (beam_data%flv))
  call state_matrix_init (state_fc)
  call state_matrix_add_state (state_fc, qn)
  call merge_state_matrices (state_hel, state_fc, state_tmp)
  call state_iterator_init (it_hel, state_hel)
  call state_iterator_init (it_tmp, state_tmp)
  do while (state_iterator_is_valid (it_hel))
     call interaction_add_state (beam%int, &
          state_iterator_get_quantum_numbers (it_tmp), &
          value=state_iterator_get_matrix_element (it_hel))
     call state_iterator_advance (it_hel)
     call state_iterator_advance (it_tmp)
```

```
      end do
      call interaction_freeze (beam%int)
      call interaction_set_momenta &
           (beam%int, beam_data%p, outgoing = .true.)
      call state_matrix_final (state_hel)
      call state_matrix_final (state_fc)
      call state_matrix_final (state_tmp)
    end subroutine beam_init
```

Finalizer:

⟨*Beams: public*⟩+≡
```
    public :: beam_final
```

⟨*Beams: procedures*⟩+≡
```
    elemental subroutine beam_final (beam)
      type(beam_t), intent(inout) :: beam
      call interaction_final (beam%int)
    end subroutine beam_final
```

I/O:

⟨*Beams: public*⟩+≡
```
    public :: beam_write
```

⟨*Beams: procedures*⟩+≡
```
    subroutine beam_write (beam, unit, verbose, show_momentum_sum, show_mass)
      type(beam_t), intent(in) :: beam
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      select case (interaction_get_n_out (beam%int))
      case (1);  write (u, *) "Decaying particle:"
      case (2);  write (u, *) "Colliding beams:"
      end select
      call interaction_write &
           (beam%int, unit, verbose, show_momentum_sum, show_mass)
    end subroutine beam_write
```

Defined assignment: deep copy

⟨*Beams: public*⟩+≡
```
    public :: assignment(=)
```

⟨*Beams: interfaces*⟩≡
```
    interface assignment(=)
       module procedure beam_assign
    end interface
```

⟨*Beams: procedures*⟩+≡
```
    subroutine beam_assign (beam_out, beam_in)
      type(beam_t), intent(out) :: beam_out
      type(beam_t), intent(in) :: beam_in
      beam_out%int = beam_in%int
    end subroutine beam_assign
```

9.1.7 Inherited procedures

⟨*Beams: public*⟩+≡
```
public :: interaction_set_source_link
```

⟨*Beams: interfaces*⟩+≡
```
interface interaction_set_source_link
   module procedure interaction_set_source_link_beam
end interface
```

⟨*Beams: procedures*⟩+≡
```
subroutine interaction_set_source_link_beam (int, i, beam1, i1)
  type(interaction_t), intent(inout) :: int
  type(beam_t), intent(in), target :: beam1
  integer, intent(in) :: i, i1
  call interaction_set_source_link (int, i, beam1%int, i1)
end subroutine interaction_set_source_link_beam
```

9.1.8 Accessing contents

Return the interaction component – as a pointer, to avoid any copying.

⟨*Beams: public*⟩+≡
```
public :: beam_get_int_ptr
```

⟨*Beams: procedures*⟩+≡
```
function beam_get_int_ptr (beam) result (int)
  type(interaction_t), pointer :: int
  type(beam_t), intent(in), target :: beam
  int => beam%int
end function beam_get_int_ptr
```

Set beam momenta directly. (Used for cascade decays.)

⟨*Beams: public*⟩+≡
```
public :: beam_set_momenta
```

⟨*Beams: procedures*⟩+≡
```
subroutine beam_set_momenta (beam, p)
  type(beam_t), intent(inout) :: beam
  type(vector4_t), dimension(:), intent(in) :: p
  call interaction_set_momenta (beam%int, p)
end subroutine beam_set_momenta
```

9.1.9 Test

⟨*Beams: public*⟩+≡
```
public :: beam_test
```

⟨*Beams: procedures*⟩+≡
```
subroutine beam_test ()
  use os_interface, only: os_data_t
  type(os_data_t) :: os_data
  type(beam_data_t), target :: beam_data
  type(beam_t) :: beam
```

```
      real(default) :: sqrts
      type(flavor_t), dimension(2) :: flv
      type(polarization_t), dimension(2) :: pol
      type(model_t), pointer :: model
      print *, "*** Read model file"
      call syntax_model_file_init ()
      call model_list_read_model &
           (var_str("QCD"), var_str("test.mdl"), os_data, model)
      call syntax_model_file_final ()
      print *
      print *, "*** Scattering"
      sqrts = 500
      call flavor_init (flv, ((/1,-1/)), model)
      call polarization_init_circular (pol(1), flv(1), 0.5_default)
      call polarization_init_transversal (pol(2), flv(2), 0._default, 1._default)
      call beam_data_init_sqrts (beam_data, sqrts, flv, pol)
      call beam_data_write (beam_data)
      print *
      call beam_init (beam, beam_data)
      call beam_write (beam)
      call beam_final (beam)
      call beam_data_final (beam_data)
      print *
      print *, "*** Decay"
      call flavor_init (flv(1), 23, model)
      call polarization_init_longitudinal (pol(1), flv(1), 0.4_default)
      call beam_data_init_decay (beam_data, flv(1:1), pol(1:1))
      call beam_data_write (beam_data)
      print *
      call beam_init (beam, beam_data)
      call beam_write (beam)
      call beam_final (beam)
      call beam_data_final (beam_data)
    end subroutine beam_test
```

Chapter 10

Spectra and structure functions

Each type of spectrum and structure function gets its own module with a common API; all modules are collected and accessed via the wrapper module `spectra`.

10.1 Tools

This module contains auxiliary procedures that can be accessed by the structure function code.

⟨`sf_aux.f90`⟩≡
  ⟨*File header*⟩

  ```
  module sf_aux
  ```

  ⟨*Use kinds*⟩
  ```
    use constants, only: twopi !NODEP!
  ```
  ⟨*Use file utils*⟩
  ```
    use lorentz !NODEP!
  ```

  ⟨*Standard module head*⟩

  ⟨*SF aux: public*⟩

  ⟨*SF aux: parameters*⟩

  ⟨*SF aux: types*⟩

  ```
  contains
  ```

  ⟨*SF aux: procedures*⟩

  ```
  end module sf_aux
  ```

10.1.1 Momentum splitting

Let us consider first an incoming parton with momentum k and invariant mass squared $s = k^2$ that splits into two partons with momenta q, p and invariant masses $t = q^2$ and $u = p^2$. (This is an abuse of the Mandelstam notation. t is actually the momentum transfer, assuming that p is radiated and q initiates the hard process.) The energy is split among the partons such that if $E = k^0$, we have $q^0 = xE$ and $p^0 = \bar{x}E$, where $\bar{x} \equiv 1 - x$.

We define the angle θ as the polar angle of p w.r.t. the momentum axis of the incoming momentum k. Ignoring azimuthal angle, we can write the four-momenta in the basis (E, p_T, p_L) as

$$
k = \begin{pmatrix} E \\ 0 \\ p \end{pmatrix}, \qquad p = \begin{pmatrix} \bar{x}E \\ \bar{x}\bar{p}\sin\theta \\ \bar{x}\bar{p}\cos\theta \end{pmatrix}, \qquad q = \begin{pmatrix} xE \\ -\bar{x}\bar{p}\sin\theta \\ p - \bar{x}\bar{p}\cos\theta \end{pmatrix}, \tag{10.1}
$$

where the first two mass-shell conditions are

$$
p^2 = E^2 - s, \qquad \bar{p}^2 = E^2 - \frac{u}{\bar{x}^2}. \tag{10.2}
$$

The second condition implies that, for positive u, $\bar{x}^2 > u/E^2$, or equivalently

$$
x < 1 - \sqrt{u}/E. \tag{10.3}
$$

We are interested in the third mass-shell conditions: s and u are fixed, so we need t as a function of $\cos\theta$:

$$
t = -2\bar{x}\left(E^2 - p\bar{p}\cos\theta\right) + s + u. \tag{10.4}
$$

Solving for $\cos\theta$, we get

$$
\cos\theta = \frac{2\bar{x}E^2 + t - s - u}{2\bar{x}p\bar{p}}. \tag{10.5}
$$

We can compute $\sin\theta$ numerically as $\sin^2\theta = 1 - \cos^2\theta$, but it is important to reexpress this in view of numerical stability. To this end, we first determine the bounds for t. The cosine must be between -1 and 1, so the bounds are

$$
t_0 = -2\bar{x}\left(E^2 + p\bar{p}\right) + s + u, \tag{10.6}
$$
$$
t_1 = -2\bar{x}\left(E^2 - p\bar{p}\right) + s + u. \tag{10.7}
$$

Computing $\sin^2\theta$ from $\cos\theta$ above, we observe that the numerator is a quadratic polynomial in t which has the zeros t_0 and t_1, while the common denominator is given by $(2\bar{x}p\bar{p})^2$. Hence, we can write

$$
\sin^2\theta = -\frac{(t - t_0)(t - t_1)}{(2\bar{x}p\bar{p})^2} \qquad \text{and} \qquad \cos\theta = \frac{(t - t_0) + (t - t_1)}{4\bar{x}p\bar{p}}, \tag{10.8}
$$

which is free of large cancellations near $t = t_0$ or $t = t_1$.

If all is massless, i.e., $s = u = 0$, this simplifies to

$$
t_0 = -4\bar{x}E^2, \qquad\qquad\qquad t_1 = 0, \tag{10.9}
$$
$$
\sin^2\theta = -\frac{t}{\bar{x}E^2}\left(1 + \frac{t}{4\bar{x}E^2}\right), \qquad \cos\theta = 1 + \frac{t}{2\bar{x}E^2}. \tag{10.10}
$$

Here is the implementation. First, we define a container for the kinematical integration limits and some further data.

⟨*SF aux: public*⟩≡
```
public :: splitting_data_t
```

⟨*SF aux: types*⟩≡
```
type :: splitting_data_t
   private
   real(default) :: x0 = 0
   real(default) :: x1
   real(default) :: t0
   real(default) :: t1
   real(default) :: phi0 = 0
   real(default) :: phi1 = twopi
   real(default) :: E, p, s, u, m2
   real(default) :: x, xb, pb
   real(default) :: t, phi
end type splitting_data_t
```

This is the initializer for the data. The input consists of the incoming momentum, its invariant mass squared, and the invariant mass squared of the radiated particle. m is the *physical* mass of the outgoing particle. The t bounds depend on the chosen x value and cannot be determined yet.

⟨*SF aux: public*⟩+≡
```
public :: new_splitting_data
```

⟨*SF aux: procedures*⟩≡
```
function new_splitting_data (k, mk2, mr2, m) result (d)
   type(splitting_data_t) :: d
   type(vector4_t), intent(in) :: k
   real(default), intent(in) :: mk2, mr2, m
   d%E = energy (k)
   d%x1 = 1 - sqrt (max (mr2, 0._default)) / d%E
   d%p = sqrt (d%E**2 - mk2)
   d%s = mk2
   d%u = mr2
   d%m2 = m**2
end function new_splitting_data
```

I/O for debugging:

⟨*SF aux: public*⟩+≡
```
public :: splitting_data_write
```

⟨*SF aux: procedures*⟩+≡
```
subroutine splitting_data_write (d, unit)
   type(splitting_data_t), intent(in) :: d
   integer, intent(in), optional :: unit
   integer :: u
   u = output_unit (unit);  if (u < 0)  return
   write (u, *) "Splitting data:"
   write (u, *) " x0   =", d%x0
   write (u, *) " x    =", d%x
   write (u, *) " xb   =", d%xb
   write (u, *) " x1   =", d%x1
```

```
      write (u, *) " t0   =", d%t0
      write (u, *) " t    =", d%t
      write (u, *) " t1   =", d%t1
      write (u, *) " phi0 =", d%phi0
      write (u, *) " phi  =", d%phi
      write (u, *) " phi1 =", d%phi1
      write (u, *) " E    =", d%E
      write (u, *) " p    =", d%p
      write (u, *) " pb   =", d%pb
      write (u, *) " s    =", d%s
      write (u, *) " u    =", d%u
      write (u, *) " m2   =", d%m2
    end subroutine splitting_data_write
```

Retrieve the x bounds, if needed for x sampling. Generating an x value is done by the caller, since this is the part that depends on the nature of the structure function.

⟨*SF aux: public*⟩+≡
```
    public :: splitting_get_x_bounds
```

⟨*SF aux: procedures*⟩+≡
```
    function splitting_get_x_bounds (d) result (x)
      real(default), dimension(2) :: x
      type(splitting_data_t), intent(in) :: d
      x = (/ d%x0, d%x1 /)
    end function splitting_get_x_bounds
```

Now set the momentum fraction and compute t_0 and t_1.

⟨*SF aux: public*⟩+≡
```
    public :: splitting_set_t_bounds
```

⟨*SF aux: procedures*⟩+≡
```
    subroutine splitting_set_t_bounds (d, x, xb)
      type(splitting_data_t), intent(inout) :: d
      real(default), intent(in) :: x, xb
      real(default) :: tp, tm
      d%x = x
      d%xb = xb
      d%pb = sqrt (max (d%E**2 - d%u / xb**2, 0._default))
      tp = -2 * xb * d%E**2 + d%s + d%u
      tm = -2 * xb * d%p * d%pb
      d%t0 = tp + tm
      d%t1 = tp - tm
    end subroutine splitting_set_t_bounds
```

These bounds may be narrowed by external cutoffs. Necessary in particular, if $m = 0$ and $t_1 = 0$.

⟨*SF aux: public*⟩+≡
```
    public :: splitting_narrow_t_bounds
```

⟨*SF aux: procedures*⟩+≡
```
    subroutine splitting_narrow_t_bounds (d, qmin, qmax)
      type(splitting_data_t), intent(inout) :: d
```

```
      real(default), intent(in), optional :: qmin, qmax
      if (present (qmax))  d%t0 = max (d%t0, - qmax ** 2)
      if (present (qmin))  d%t1 = min (d%t1, - qmin ** 2)
    end subroutine splitting_narrow_t_bounds
```

Compute a value for the momentum transfer t, using a random number r. We assume a logarithmic distribution for $t - m^2$, corresponding to the propagator $1/(t - m^2)$ with the physical mass m for the outgoing particle. Optionally, we can narrow the kinematical bounds.

⟨*SF aux: public*⟩+≡
```
    public :: splitting_sample_t
```

⟨*SF aux: procedures*⟩+≡
```
    subroutine splitting_sample_t (d, r, t0, t1)
      type(splitting_data_t), intent(inout) :: d
      real(default), intent(in) :: r
      real(default), intent(in), optional :: t0, t1
      real(default) :: tt0, tt1
      tt0 = d%t0;  if (present (t0))  tt0 = max (t0, tt0)
      tt1 = d%t1;  if (present (t1))  tt1 = min (t1, tt1)
      d%t = d%m2 + (tt0 - d%m2) * exp (r * log ((tt1 - d%m2) / (tt0 - d%m2)))
    end subroutine splitting_sample_t
```

State that we can ignore recoil. The momentum transfer t is set to its upper limit (which is zero for massless particles).

⟨*SF aux: public*⟩+≡
```
    public :: splitting_set_collinear
```

⟨*SF aux: procedures*⟩+≡
```
    subroutine splitting_set_collinear (d)
      type(splitting_data_t), intent(inout) :: d
      d%t = d%t1
    end subroutine splitting_set_collinear
```

This is trivial, but provided for convenience:

⟨*SF aux: public*⟩+≡
```
    public :: splitting_sample_phi
```

⟨*SF aux: procedures*⟩+≡
```
    subroutine splitting_sample_phi (d, r)
      type(splitting_data_t), intent(inout) :: d
      real(default), intent(in) :: r
      d%phi = (1-r) * d%phi0 + r * d%phi1
    end subroutine splitting_sample_phi
```

In this function, we actually perform the splitting. Apart from the splitting data, we need the incoming momentum k, the momentum transfer t, and the azimuthal angle ϕ. The momentum fraction x is already known here.

Alternatively, we can split without recoil. The azimuthal angle is irrelevant, and the momentum transfer is always equal to the upper limit t_1, so the polar angle is zero. Still, we have to account for nonvanishing invariant masses.

⟨*SF aux: public*⟩+≡
```
    public :: split_momentum
```

⟨*SF aux: procedures*⟩+≡
```
    function split_momentum (k, d) result (q)
      type(vector4_t), dimension(2) :: q
      type(vector4_t), intent(in) :: k
      type(splitting_data_t), intent(in) :: d
      real(default) :: ct, st, cp, sp
      type(lorentz_transformation_t) :: rot
      real(default) :: tt0, tt1, den
      type(vector3_t) :: kk, q1, q2
      if (d%t < d%t1) then
         tt0 = d%t - d%t0
         tt1 = d%t - d%t1
         den = 2 * d%xb * d%p * d%pb
         ct = (tt0 + tt1) / (2 * den)
         st = - (tt0 * tt1) / den**2
         cp = cos (d%phi)
         sp = sin (d%phi)
         rot = rotation_to_2nd (3, space_part (k))
         q1 = vector3_moving (d%xb * d%pb * (/ st * cp, st * sp, ct /))
         q2 = vector3_moving (d%p, 3) - q1
         q(1) = rot * vector4_moving (d%xb * d%E, q1)
         q(2) = rot * vector4_moving (d%x * d%E, q2)
      else if (d%s /= 0 .or. d%u /= 0) then
         kk = space_part (k)
         q1 = d%xb * (d%pb / d%p) * kk
         q2 = kk - q1
         q(1) = vector4_moving (d%xb * d%E, q1)
         q(2) = vector4_moving (d%x * d%E, q2)
      else
         q(1) = d%xb * k
         q(2) = d%x * k
      end if
    end function split_momentum
```

10.1.2   Mass-shell projection

Momenta generated by splitting will in general be off-shell. They are on-shell
only if they are collinear and massless. This subroutine puts them on shell by
brute force, violating either momentum or energy conservation. The direction
of three-momentum is always retained.

⟨*SF aux: parameters*⟩≡
```
    integer, parameter, public :: KEEP_ENERGY = 0, KEEP_MOMENTUM = 1
```

⟨*SF aux: public*⟩+≡
```
    public :: on_shell
```

⟨*SF aux: procedures*⟩+≡
```
    elemental subroutine on_shell (p, mass, keep)
      type(vector4_t), intent(inout) :: p
      real(default), intent(in) :: mass
      integer, intent(in) :: keep
      real(default) :: E, pn
      select case (keep)
```

```
   case (KEEP_ENERGY)
      E = energy (p)
      pn = sqrt (max (E**2 - mass**2, 0._default))
      p = vector4_moving (E, pn * direction (space_part (p)))
   case (KEEP_MOMENTUM)
      E = sqrt (space_part (p) ** 2 + mass **2)
      p = vector4_moving (E, space_part (p))
   end select
end subroutine on_shell
```

10.2 Photon radiation: ISR

⟨sf_isr.f90⟩≡
  ⟨*File header*⟩

  module sf_isr

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
    use constants, only: pi !NODEP!
  ⟨*Use file utils*⟩
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use sm_physics, only: Li2 !NODEP!
    use models
    use flavors
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use sf_aux

  ⟨*Standard module head*⟩

  ⟨*ISR: public*⟩

  ⟨*ISR: parameters*⟩

  ⟨*ISR: types*⟩

  contains

  ⟨*ISR: procedures*⟩

  end module sf_isr

10.2.1 Physics

The ISR structure function is in the most crude approximation (LLA without α corrections, i.e. ϵ^0)

$$f_0(x) = \epsilon(1-x)^{-1+\epsilon} \qquad \text{with} \qquad \epsilon = \frac{\alpha}{\pi}q_e^2 \ln\frac{s}{m^2}, \qquad (10.11)$$

where m is the mass of the incoming (and outgoing) particle, which is initially assumed on-shell.

Here, the form of ϵ results from the kinematical bounds for the momentum squared of the outgoing particle, which in the limit $m^2 \ll s$ are given by

$$t_0 = -2\bar{x}E(E+p) + m^2 \approx -\bar{x}s, \qquad (10.12)$$

$$t_1 = -2\bar{x}E(E-p) + m^2 \approx xm^2, \qquad (10.13)$$

so the integration over the propagator $1/(t - m^2)$ yields

$$\ln \frac{t_0 - m^2}{t_1 - m^2} = \ln \frac{s}{m^2}. \tag{10.14}$$

In $f_0(x)$, there is an integrable singularity at $x = 1$ which does not spoil the integration, but would lead to an unbounded $f_{\max}$. Therefore, we map this singularity like

$$x = 1 - (1 - x')^{1/\epsilon} \tag{10.15}$$

such that

$$\int dx\, f_0(x) = \int dx' \tag{10.16}$$

The structure function has three parameters: α, m_{in} of the incoming particle and s, the hard scale. Internally, we store the exponent ϵ which is the relevant parameter. (In conventional notation, $\epsilon = \beta/2$.) As defaults, we take the actual values of α (which is probably $\alpha(s)$), the actual mass m_{in} and the squared total c.m. energy s.

Including ϵ, ϵ^2, and ϵ^3 corrections, the successive approximation of the ISR structure function read

$$f_0(x) = \epsilon(1 - x)^{-1+\epsilon} \tag{10.17}$$

$$f_1(x) = g_1(\epsilon)\, f_0(x) - \frac{\epsilon}{2}(1 + x) \tag{10.18}$$

$$f_2(x) = g_2(\epsilon)\, f_0(x) - \frac{\epsilon}{2}(1 + x)$$
$$- \frac{\epsilon^2}{8}\left(\frac{1 + 3x^2}{1 - x}\ln x + 4(1 + x)\ln(1 - x) + 5 + x\right) \tag{10.19}$$

$$f_3(x) = g_3(\epsilon)\, f_0(x) - \frac{\epsilon}{2}(1 + x)$$
$$- \frac{\epsilon^2}{8}\left(\frac{1 + 3x^2}{1 - x}\ln x + 4(1 + x)\ln(1 - x) + 5 + x\right)$$
$$- \frac{\epsilon^3}{48}\left((1 + x)\left[6\,\mathrm{Li}_2(x) + 12\ln^2(1 - x) - 3\pi^2\right] + 6(x + 5)\ln(1 - x)\right.$$
$$+ \frac{1}{1 - x}\left[\frac{3}{2}(1 + 8x + 3x^2)\ln x + 12(1 + x^2)\ln x \ln(1 - x)\right.$$
$$\left.\left. - \frac{1}{2}(1 + 7x^2)\ln^2 x + \frac{1}{4}(39 - 24x - 15x^2)\right]\right) \tag{10.20}$$

where the successive approximations to the prefactor of the leading singularity

$$g(\epsilon) = \frac{\exp\left(\epsilon(-\gamma_E + \frac{3}{4})\right)}{\Gamma(1 + \epsilon)}, \tag{10.21}$$

are given by

$$g_0(\epsilon) = 1 \tag{10.22}$$

$$g_1(\epsilon) = 1 + \frac{3}{4}\epsilon \tag{10.23}$$

$$g_2(\epsilon) = 1 + \frac{3}{4}\epsilon + \frac{27 - 8\pi^2}{96}\epsilon^2 \tag{10.24}$$

$$g_3(\epsilon) = 1 + \frac{3}{4}\epsilon + \frac{27 - 8\pi^2}{96}\epsilon^2 + \frac{27 - 24\pi^2 + 128\zeta(3)}{384}\epsilon^3, \tag{10.25}$$

where, numerically

$$\zeta(3) = 1.2020569031595942853997381615 1\ldots \tag{10.26}$$

Although one could calculate the function $g(\epsilon)$ exactly, truncating its Taylor expansion ensures the exact normalization of the truncated structure function at each given order:

$$\int_0^1 dx\, f_i(x) = 1 \qquad \text{for all } i. \tag{10.27}$$

Effectively, the $O(\epsilon)$ correction reduces the low-x tail of the structure function by 50% while increasing the coefficient of the singularity by $O(\epsilon)$. Relative to this, the $O(\epsilon^2)$ correction slightly enhances $x > \frac{1}{2}$ compared to $x < \frac{1}{2}$. At $x = 0$, $f_2(x)$ introduces a logarithmic singularity which should be cut off at $x_0 = O(e^{-1/\epsilon})$: for lower x the perturbative series breaks down. The f_3 correction is slightly positive for low x values and negative near $x = 1$, where the Li_2 piece slightly softens the singularity at $x = 1$.

Instead of the definition for ϵ given above, it is customary to include a universal nonlogarithmic piece:

$$\epsilon = \frac{\alpha}{\pi} q_e^2 \left(\ln \frac{s}{m^2} - 1 \right) \tag{10.28}$$

10.2.2 Implementation

In the concrete implementation, the zeroth order mapping (10.15) is implemented, and the Jacobian is equal to $f_i(x)/f_0(x)$. This can be written as

$$\frac{f_0(x)}{f_0(x)} = 1 \tag{10.29}$$

$$\frac{f_1(x)}{f_0(x)} = 1 + \frac{3}{4}\epsilon - \frac{1 - x^2}{2(1 - x')} \tag{10.30}$$

$$\frac{f_2(x)}{f_0(x)} = 1 + \frac{3}{4}\epsilon + \frac{27 - 8\pi^2}{96}\epsilon^2 - \frac{1 - x^2}{2(1 - x')}$$
$$- \frac{(1 + 3x^2)\ln x + (1 - x)\left(4(1 + x)\ln(1 - x) + 5 + x\right)}{8(1 - x')}\epsilon \tag{10.31}$$

For $x = 1$ (i.e., numerically indistinguishable from 1), this reduces to

$$\frac{f_0(x)}{f_0(x)} = 1 \tag{10.32}$$

$$\frac{f_1(x)}{f_0(x)} = 1 + \frac{3}{4}\epsilon \tag{10.33}$$

$$\frac{f_2(x)}{f_0(x)} = 1 + \frac{3}{4}\epsilon + \frac{27 - 8\pi^2}{96}\epsilon^2 \tag{10.34}$$

The last line in (10.31) is zero for

$$x_{\min} = 0.0071405332973459283954981027 5603 \tag{10.35}$$

(Mathematica result), independent of ϵ. For x values less than this we ignore this correction because of the logarithmic singularity which should in principle be resummed.

10.2.3 The ISR data block

⟨*ISR: public*⟩≡
```
public :: isr_data_t
```

⟨*ISR: types*⟩≡
```
type :: isr_data_t
  private
  type(model_t), pointer :: model => null ()
  type(flavor_t) :: flv
  real(default) :: alpha = 0
  real(default) :: q_max = 0
  real(default) :: real_mass = 0
  real(default) :: mass = 0
  real(default) :: eps = 0
  real(default) :: log = 0
  integer :: order = 3
  integer :: error = NONE
end type isr_data_t
```

Error codes
⟨*ISR: parameters*⟩≡
```
integer, parameter :: NONE = 0
integer, parameter :: ZERO_MASS = 1
integer, parameter :: Q_MAX_TOO_SMALL = 2
integer, parameter :: EPS_TOO_LARGE = 3
integer, parameter :: INVALID_ORDER = 4
```
Generate flavor-dependent ISR data:
⟨*ISR: public*⟩+≡
```
public :: isr_data_init
```

⟨*ISR: procedures*⟩≡
```
subroutine isr_data_init (data, model, flv, alpha, q_max, mass)
  type(isr_data_t), intent(out) :: data
  type(model_t), intent(in), target :: model
  type(flavor_t), intent(in) :: flv
```

```
      real(default), intent(in) :: alpha
      real(default), intent(in) :: q_max
      real(default), intent(in), optional :: mass
      data%model => model
      data%flv = flv
      data%alpha = alpha
      data%q_max = q_max
      data%real_mass = flavor_get_mass (flv)
      select case (char (model_get_name (data%model)))
      case ("QCD","Test")
         call msg_fatal ("ISR structure function not available for model " &
               // char (model_get_name (data%model)) // ".")
      end select
      if (present (mass)) then
         if (mass > 0) then
            data%mass = mass
         else
            data%mass = data%real_mass
         end if
      else
         data%mass = data%real_mass
      end if
      if (data%mass == 0) then
         data%error = ZERO_MASS;  return
      else if (data%mass >= data%q_max) then
         data%error = Q_MAX_TOO_SMALL;  return
      end if
      data%log = log (1 + (data%q_max / data%mass)**2)
      data%eps = data%alpha / pi &
            * flavor_get_charge (data%flv)**2 &
            * (2 * log (data%q_max / data%mass) - 1)
      if (data%eps > 1) then
         data%error = EPS_TOO_LARGE;  return
      end if
   end subroutine isr_data_init
```

Explicitly set ISR order
⟨*ISR: public*⟩+≡
```
    public :: isr_data_set_order
```
⟨*ISR: procedures*⟩+≡
```
    elemental subroutine isr_data_set_order (data, order)
      type(isr_data_t), intent(inout) :: data
      integer, intent(in) :: order
      if (order < 0 .or. order > 3) then
         data%error = INVALID_ORDER
      else
         data%order = order
      end if
    end subroutine isr_data_set_order
```

Handle error conditions. Should always be done after initialization, unless we are sure everything is ok.
⟨*ISR: public*⟩+≡

```
        public :: isr_data_check
```

⟨*ISR: procedures*⟩+≡
```
    subroutine isr_data_check (data)
      type(isr_data_t), intent(in) :: data
      select case (data%error)
      case (ZERO_MASS)
         call msg_fatal (" ISR: Particle mass is zero")
      case (Q_MAX_TOO_SMALL)
         call msg_fatal (" ISR: Particle mass exceeds Qmax")
      case (EPS_TOO_LARGE)
         call msg_fatal (" ISR: Expansion parameter too large, perturbative expansion breaks down")
      case (INVALID_ORDER)
         call msg_error (" ISR: LLA order invalid (valid values are 0,1,2,3)")
      end select
    end subroutine isr_data_check
```

Output

⟨*ISR: public*⟩+≡
```
    public :: isr_data_write
```

⟨*ISR: procedures*⟩+≡
```
    subroutine isr_data_write (data, unit)
      type(isr_data_t), intent(in) :: data
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, *) "ISR data:"
      write (u, *) "  prt   = ", char (flavor_get_name (data%flv))
      write (u, *) "  alpha = ", data%alpha
      write (u, *) "  q_max = ", data%q_max
      write (u, *) "  mass  = ", data%mass
      write (u, *) "  eps   = ", data%eps
      write (u, *) "  log   = ", data%log
      write (u, *) "  order = ", data%order
    end subroutine isr_data_write
```

10.2.4 The ISR object

The `isr_t` data type is a $1 \rightarrow 2$ interaction, i.e., we allow for single-photon emission only (but use the multi-photon resummed radiator function). The particles are ordered as (incoming, photon, outgoing).

There is no need to handle several flavors (and data blocks) in parallel, since ISR is always applied immediately after beam collision. (ISR for partons is accounted for by the PDFs themselves.) Polarization is carried through, i.e., we retain the polarization of the incoming particle and treat the emitted photon as unpolarized. Color is trivially carried through. This implies that particles 1 and 3 should be locked together.

⟨*ISR: public*⟩+≡
```
    public :: interaction_init_isr
```

⟨*ISR: procedures*⟩+≡

```
subroutine interaction_init_isr (int, data)
  type(interaction_t), intent(out) :: int
  type(isr_data_t), intent(in) :: data
  type(quantum_numbers_mask_t), dimension(3) :: mask
  integer, dimension(3) :: lock
  type(polarization_t) :: pol
  type(quantum_numbers_t), dimension(1) :: qn_fc, qn_hel
  type(flavor_t) :: flv_photon
  type(quantum_numbers_t) :: qn_photon, qn
  type(state_iterator_t) :: it_hel
  mask = new_quantum_numbers_mask (.false., .false., &
      mask_h = (/ .false., .true., .false. /))
  lock = (/ 3, 0, 1 /)
  call interaction_init &
      (int, 1, 0, 2, mask=mask, lock=lock, set_relations=.true.)
  call flavor_init (flv_photon, PHOTON, data%model)
  call quantum_numbers_init (qn_photon, flv_photon)
  call polarization_init_generic (pol, data%flv)
  call quantum_numbers_init (qn_fc(1), &
      flv = data%flv, col = color_from_flavor (data%flv))
  call state_iterator_init (it_hel, pol%state)
  do while (state_iterator_is_valid (it_hel))
     qn_hel = state_iterator_get_quantum_numbers (it_hel)
     qn = qn_hel(1) .merge. qn_fc(1)
     call interaction_add_state (int, (/ qn, qn_photon, qn /))
     call state_iterator_advance (it_hel)
  end do
  call polarization_final (pol)
  call interaction_freeze (int)
end subroutine interaction_init_isr
```

10.2.5 ISR structure function

The ISR structure function allows for a straightforward mapping of the unit interval. So, to leading order, the structure function value is unity, but the x value is transformed. Higher orders affect the function value.

The structure function implementation applies the above mapping to the input (random) number `r` to generate the momentum fraction `x` and the function value `f`. For numerical stability reasons, we also output `xb`, which is $\bar{x} = 1 - x$. The mapping ensures that $f = 1$ at leading order, and the higher-order corrections are well-behaved.

⟨*ISR: procedures*⟩+≡

```
subroutine strfun (f, x, xb, r, data)
  real(default), intent(out) :: f, x, xb
  real(default), intent(in) :: r
  type(isr_data_t), intent(in) :: data
  real(default) :: eps
  real(default) :: rb, log_x, log_xb, x_2
  real(default), parameter :: &
      & xmin = 0.0071405332973459283954981027560 3_default
  real(default), parameter :: &
```

```
      & zeta3 = 1.202056903159594285399738161151_default
   real(default), parameter :: &
      & g1 = 3._default / 4._default, &
      & g2 = (27 - 8*pi**2) / 96._default, &
      & g3 = (27 - 24*pi**2 + 128*zeta3) / 384._default
   eps = data%eps
   rb = 1 - r
   if (rb < tiny(1._default)**eps) then
      xb = 0
   else
      xb = rb**(1/eps)
   end if
   x = 1 - xb
   if (data%order > 0) then
      f = 1 + g1 * eps
      x_2 = x*x
      if (rb>0)  f = f - (1-x_2) / (2 * rb)
      if (data%order > 1) then
         f = f + g2 * eps**2
         if (rb>0 .and. xb>0 .and. x>xmin) then
            log_x = log (x)
            log_xb = log (xb)
            f = f - ((1+3*x_2)*log_x + xb * (4*(1+x)*log_xb + 5 + x)) &
                  & / ( 8 * rb) * eps
         end if
         if (data%order > 2) then
            f = f + g3 * eps**3
            if (rb > 0 .and. xb > 0 .and. x > xmin) then
               f = f - ((1+x) * xb &
                        * (6 * Li2(x) + 12 * log_xb**2 - 3 * pi**2) &
                        + 1.5_default * (1 + 8*x + 3*x_2) * log_x &
                        + 6 * (x+5) * xb * log_xb &
                        + 12 * (1+x_2) * log_x * log_xb &
                        - (1 + 7*x_2) * log_x**2 / 2 &
                        + (39 - 24*x - 15*x_2) / 4) &
                        / ( 48 * rb) * eps**2
            end if
         end if
      end if
   else
      f = 1
   end if
 end subroutine strfun
```

10.2.6   ISR application

For ISR, we can compute kinematics and function value in a single step. This function works on a single beam, assuming that the input momentum has been set. We need four random numbers as input: one for x, one for Q^2, and two for the polar and azimuthal angles. Alternatively, we can skip p_T generation; in this case, we only need one.

After splitting momenta, we set the outgoing momenta on-shell. We choose

to conserve momentum, so energy conservation may be violated.

⟨*ISR: public*⟩+≡
```
    public :: interaction_apply_isr
```

⟨*ISR: procedures*⟩+≡
```
    subroutine interaction_apply_isr (int, r, isr_data)
      type(interaction_t), intent(inout) :: int
      real(default), dimension(:), intent(in) :: r
      type(isr_data_t), intent(in) :: isr_data
      type(vector4_t) :: k
      type(splitting_data_t) :: sd
      real(default) :: f, x, xb
      k = interaction_get_momentum (int, 1)
      sd = new_splitting_data (k, k**2, 0._default, isr_data%mass)
      call strfun (f, x, xb, r(1), isr_data)
      call interaction_set_matrix_element (int, cmplx (f, kind=default))
      call splitting_set_t_bounds (sd, x, xb)
      select case (size (r))
      case (1)
         call splitting_set_collinear (sd)
      case (3)
         call splitting_sample_t (sd, r(2))
         call splitting_sample_phi (sd, r(3))
      case default
         print *, "n_rand = ", size (r)
         call msg_bug (" ISR: number of random numbers must be 1 or 3")
      end select
      call interaction_set_momenta &
           (int, split_momentum (k, sd), outgoing=.true.)
    end subroutine interaction_apply_isr
```

10.3 EPA

⟨sf_epa.f90⟩≡
  ⟨File header⟩

  module sf_epa

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use constants, only: pi !NODEP!
  ⟨Use file utils⟩
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use models
    use flavors
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use sf_aux

  ⟨Standard module head⟩

  ⟨EPA: public⟩

  ⟨EPA: parameters⟩

  ⟨EPA: types⟩

  contains

  ⟨EPA: procedures⟩

  end module sf_epa

10.3.1 Physics

The EPA structure function for a photon inside an (elementary) particle p with energy E, mass m and charge q_p (e.g., electron) is given by ($\bar{x} \equiv 1 - x$)

$$f(x) = \frac{\alpha}{\pi} q_p^2 \frac{1}{x} \left[\left(\bar{x} + \frac{x^2}{2} \right) \ln \frac{Q_{\max}^2}{Q_{\min}^2} - \left(1 - \frac{x}{2} \right)^2 \ln \frac{x^2 + \frac{Q_{\max}^2}{E^2}}{x^2 + \frac{Q_{\min}^2}{E^2}} - x^2 \frac{m^2}{Q_{\min}^2} \left(1 - \frac{Q_{\min}^2}{Q_{\max}^2} \right) \right].$$
$$(10.36)$$

If no explicit Q bounds are provided, the kinematical bounds are

$$-Q_{\max}^2 = t_0 = -2\bar{x}(E^2 + p\bar{p}) + 2m^2 \approx -4\bar{x}E^2, \qquad (10.37)$$

$$-Q_{\min}^2 = t_1 = -2\bar{x}(E^2 - p\bar{p}) + 2m^2 \approx -\frac{x^2}{\bar{x}}m^2. \qquad (10.38)$$

The second and third terms in (10.36) are negative definite (and subleading). Noting that $\bar{x} + x^2/2$ is bounded between $1/2$ and 1, we derive that $f(x)$ is always

smaller than

$$\bar{f}(x) = \frac{\alpha}{\pi}\, q_p^2\, \frac{L - 2\ln x}{x} \qquad \text{where} \qquad L = \ln \frac{\min(4E_{\max}^2, Q_{\max}^2)}{\max(m^2, Q_{\min}^2)}, \qquad (10.39)$$

where we allow for explicit Q bounds that narrow the kinematical range. Therefore, we generate this distribution:

$$\int_{x_0}^{x_1} dx\, \bar{f}(x) = C(x_0, x_1) \int_0^1 dx' \qquad (10.40)$$

We set

$$\ln x = \frac{1}{2}\left\{ L - \sqrt{L^2 - 4\left[x'\ln x_1(L - \ln x_1) + \bar{x}'\ln x_0(L - \ln x_0)\right]} \right\} \qquad (10.41)$$

such that $x(0) = x_0$ and $x(1) = x_1$ and

$$\frac{dx}{dx'} = \left(\frac{\alpha}{\pi}\right)^{-1} x \frac{C(x_0, x_1)}{L - 2\ln x} \qquad (10.42)$$

with

$$C(x_0, x_1) = \frac{\alpha}{\pi}q_p^2\left[\ln x_1(L - \ln x_1) - \ln x_0(L - \ln x_0)\right] \qquad (10.43)$$

such that (10.40) is satisfied. Finally, we have

$$\int_{x_0}^{x_1} dx\, f(x) = C(x_0, x_1) \int_0^1 dx'\, \frac{f(x(x'))}{\bar{f}(x(x'))} \qquad (10.44)$$

where x' is calculated from x via (10.41)

10.3.2   The EPA data block

The EPA parameters are: α, $E_{\max}$, m, $Q_{\min}$, and $x_{\min}$. Instead of m we can use the incoming particle PDG code as input; from this we can deduce the mass and charge.

Internally we store in addition $C_{0/1} = \frac{\alpha}{\pi}q_e^2 \ln x_{0/1}(L - \ln x_{0/1})$, the c.m. energy squared and the incoming particle mass.

⟨*EPA: public*⟩≡
```
  public :: epa_data_t
```
⟨*EPA: types*⟩≡
```
  type :: epa_data_t
     private
     type(model_t), pointer :: model => null ()
     type(flavor_t) :: flv
     real(default) :: alpha
     real(default) :: x_min
     real(default) :: x_max
     real(default) :: q_min
     real(default) :: q_max
     real(default) :: E_max
     real(default) :: mass
     real(default) :: log
     real(default) :: c0
     real(default) :: c1
     integer :: error = 0
  end type epa_data_t
```

785

Error codes

⟨*EPA: parameters*⟩≡
```
    integer, parameter :: NONE = 0
    integer, parameter :: ZERO_QMIN = 1
    integer, parameter :: Q_MAX_TOO_SMALL = 2
    integer, parameter :: ZERO_XMIN = 3
```
⟨*EPA: public*⟩+≡
```
    public :: epa_data_init
```
⟨*EPA: procedures*⟩≡
```
    subroutine epa_data_init (data, model, flv, alpha, x_min, q_min, E_max, mass)
      type(epa_data_t), intent(inout) :: data
      type(model_t), intent(in), target :: model
      type(flavor_t), intent(in) :: flv
      real(default), intent(in) :: alpha, x_min, q_min, E_max
      real(default), intent(in), optional :: mass
      data%model => model
      data%flv = flv
      data%alpha = alpha
      data%E_max = E_max
      data%x_min = x_min
      data%x_max = 1
      if (data%x_min == 0) then
         data%error = ZERO_XMIN;   return
      end if
      data%q_min = q_min
      data%q_max = 2 * data%E_max
      select case (char (model_get_name (data%model)))
      case ("QCD","Test")
         call msg_fatal ("EPA structure function not available for model " &
               // char (model_get_name (data%model)) // ".")
      end select
      if (present (mass)) then
         data%mass = mass
      else
         data%mass = flavor_get_mass (flv)
      end if
      if (max (data%mass, data%q_min) == 0) then
         data%error = ZERO_QMIN;   return
      else if (max (data%mass, data%q_min) >= data%E_max) then
         data%error = Q_MAX_TOO_SMALL;  return
      end if
      data%log = log (4 * (data%E_max / max (data%mass, data%q_min)) ** 2 )
      data%c0 = data%alpha / pi &
           * flavor_get_charge (data%flv)**2 &
           * log (data%x_min) * (data%log - log (data%x_min))
      data%c1 = data%alpha / pi &
           * flavor_get_charge (data%flv)**2 &
           * log (data%x_max) * (data%log - log (data%x_max))
    end subroutine epa_data_init
```

Handle error conditions. Should always be done after initialization, unless we are sure everything is ok.

⟨*EPA: public*⟩+≡

```
      public :: epa_data_check
```

⟨EPA: procedures⟩+≡
```
    subroutine epa_data_check (data)
      type(epa_data_t), intent(in) :: data
      select case (data%error)
      case (ZERO_QMIN)
         call msg_fatal (" EPA: Particle mass is zero")
      case (Q_MAX_TOO_SMALL)
         call msg_fatal (" EPA: Particle mass exceeds Qmax")
      case (ZERO_XMIN)
         call msg_fatal (" EPA: x_min must be larger than zero")
      end select
    end subroutine epa_data_check
```

Output

⟨EPA: public⟩+≡
```
    public :: epa_data_write
```

⟨EPA: procedures⟩+≡
```
    subroutine epa_data_write (data, unit)
      type(epa_data_t), intent(in) :: data
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, *) "EPA data:"
      write (u, *) "  prt   = ", char (flavor_get_name (data%flv))
      write (u, *) "  alpha = ", data%alpha
      write (u, *) "  x_min = ", data%x_min
      write (u, *) "  x_max = ", data%x_max
      write (u, *) "  q_min = ", data%q_min
      write (u, *) "  q_max = ", data%q_max
      write (u, *) "  E_max = ", data%q_max
      write (u, *) "  mass  = ", data%mass
      write (u, *) "  log   = ", data%log
      write (u, *) "  c0    = ", data%c0
      write (u, *) "  c1    = ", data%c1
    end subroutine epa_data_write
```

10.3.3   The EPA object

The `epa_t` data type is a $1 \to 2$ interaction. We should be able to handle several flavors in parallel, since EPA is not necessarily applied immediately after beam collision: Photons may be radiated from quarks. In that case, the partons are massless and $q_{\min}$ applies instead, so we do not need to generate several kinematical configurations in parallel.

The particles are ordered as (incoming, radiated, photon), where the photon initiates the hard interaction.

We generate an unpolarized photon and transfer initial polarization to the radiated parton. Color is transferred in the same way.

⟨EPA: public⟩+≡
```
    public :: interaction_init_epa
```

⟨EPA: procedures⟩+≡

```
subroutine interaction_init_epa (int, data)
  type(interaction_t), intent(out) :: int
  type(epa_data_t), intent(in) :: data
  type(quantum_numbers_mask_t), dimension(3) :: mask
  integer, dimension(3) :: lock
  type(polarization_t) :: pol
  type(quantum_numbers_t), dimension(1) :: qn_fc, qn_hel
  type(flavor_t) :: flv_photon
  type(quantum_numbers_t) :: qn_photon, qn
  type(state_iterator_t) :: it_hel
  integer :: i
  mask = new_quantum_numbers_mask (.false., .false., &
      mask_h = (/ .false., .false., .true. /))
  lock = (/ 2, 1, 0 /)
  call interaction_init &
      (int, 1, 0, 2, mask=mask, lock=lock, set_relations=.true.)
  call flavor_init (flv_photon, PHOTON, data%model)
  call quantum_numbers_init (qn_photon, flv_photon)
  call polarization_init_generic (pol, data%flv)
  call quantum_numbers_init (qn_fc(1), &
      flv = data%flv, col = color_from_flavor (data%flv))
  call state_iterator_init (it_hel, pol%state)
  do while (state_iterator_is_valid (it_hel))
     qn_hel = state_iterator_get_quantum_numbers (it_hel)
     qn = qn_hel(1) .merge. qn_fc(1)
     call interaction_add_state (int, (/ qn, qn, qn_photon /))
     call state_iterator_advance (it_hel)
  end do
  call polarization_final (pol)
  call interaction_freeze (int)
end subroutine interaction_init_epa
```

10.3.4   EPA structure function

The EPA structure function allows for a straightforward mapping of the unit interval. So, to leading order, the structure function value is unity, but the x value is transformed. Higher orders affect the function value.

The structure function implementation applies the above mapping to the input (random) number `r` to generate the momentum fraction `x` and the function value `f`. For numerical stability reasons, we also output `xb`, which is $\bar{x} = 1 - x$. The mapping ensures that $f = 1$ at leading order, and the higher-order corrections are well-behaved.

⟨EPA: procedures⟩+≡

```
elemental subroutine strfun (f, x, xb, r, E, data)
  real(default), intent(out) :: f, x, xb
  real(default), intent(in) :: r, E
  type(epa_data_t), intent(in) :: data
  real(default) :: rb, lx0, lx1, lx, d, den
  real(default) :: qmaxsq, qminsq
  f = 0
  rb = 1 - r
```

```
        lx0 = log (data%x_min)
        lx1 = log (data%x_max)
        d = data%log ** 2 &
             -  4 * (r * lx1 * (data%log - lx1) + rb * lx0 * (data%log - lx0))
        if (d <= 0) then
           return
        else
           lx = (data%log - sqrt (d)) / 2
        end if
        x = exp (lx)
        xb = 1 - x
        den = data%log - 2 * lx
        if (den <= 0)  return
        qminsq = max (x ** 2 / xb * data%mass ** 2, data%q_min ** 2)
        qmaxsq = min (4 * E ** 2, data%q_max ** 2)
        if (qminsq < qmaxsq) then
           f = ((xb + x ** 2 / 2) * log (qmaxsq / qminsq) &
                - (1 - x / 2) ** 2 &
                  * log ((x**2 + qmaxsq / E ** 2) / (x**2 + qminsq / E ** 2)) &
                - x ** 2 * data%mass ** 2 / qminsq * (1 - qminsq / qmaxsq)) &
              * ( data%c1 - data%c0 ) / den
        end if
      end subroutine strfun
```

10.3.5   EPA application

For EPA, we can compute kinematics and function value in a single step. This function works on a single beam, assuming that the input momentum has been set. We need four random numbers as input: one for x, one for Q^2, and two for the polar and azimuthal angles. Alternatively, we can skip p_T generation; in this case, we only need one.

For obtaining splitting kinematics, we rely on the assumption that all in-particles are mass-degenerate (or there is only one), so the generated x values are identical.

⟨*EPA: public*⟩+≡
```
    public :: interaction_apply_epa
```

⟨*EPA: procedures*⟩+≡
```
    subroutine interaction_apply_epa (int, r, epa_data)
      type(interaction_t), intent(inout) :: int
      real(default), dimension(:), intent(in) :: r
      type(epa_data_t), dimension(:), intent(in) :: epa_data
      type(vector4_t) :: k
      type(splitting_data_t) :: sd
      real(default), dimension(size(epa_data)) :: f, x, xb
      k = interaction_get_momentum (int, 1)
      sd = new_splitting_data (k, k**2, epa_data(1)%mass**2, 0._default)
      call strfun (f, x, xb, r(1), energy (k), epa_data)
      call interaction_set_flavored_values &
           (int, cmplx (f, kind=default), epa_data%flv, 2)
      call splitting_set_t_bounds (sd, x(1), xb(1))
      call splitting_narrow_t_bounds (sd, epa_data(1)%q_min, epa_data(1)%q_max)
```

```
      select case (size (r))
      case (1)
         call splitting_set_collinear (sd)
      case (3)
         call splitting_sample_t (sd, r(2))
         call splitting_sample_phi (sd, r(3))
      case default
         print *, "n_rand = ", size (r)
         call msg_bug (" EPA: number of random numbers must be 1 or 3")
      end select
      call interaction_set_momenta &
           (int, split_momentum (k, sd), outgoing=.true.)
  end subroutine interaction_apply_epa
```

10.4 EWA

⟨sf_ewa.f90⟩≡
  ⟨File header⟩

  module sf_ewa

  ⟨Use kinds⟩
  ⟨Use strings⟩
  ⟨Use file utils⟩
    use constants, only: pi !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use models
    use flavors
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use sf_aux

  ⟨Standard module head⟩

  ⟨EWA: public⟩

  ⟨EWA: parameters⟩

  ⟨EWA: types⟩

  contains

  ⟨EWA: procedures⟩

  end module sf_ewa

10.4.1 Physics

The EWA structure function for a Z or W inside a fermion (lepton or quark) depends on the vector-boson polarization. We distinguish transversal ($\pm$) and longitudinal (0) polarization.

$$F_+(x) = \frac{1}{16\pi^2} \frac{(v-a)^2 + (v+a)^2 \bar{x}^2}{x} \left[\ln\left(\frac{p_{\perp,\max}^2 + \bar{x} M^2}{\bar{x} M^2} \right) - \frac{p_{\perp,\max}^2}{p_{\perp,\max}^2 + \bar{x} M^2} \right]$$
$$(10.45)$$

$$F_-(x) = \frac{1}{16\pi^2} \frac{(v+a)^2 + (v-a)^2 \bar{x}^2}{x} \left[\ln\left(\frac{p_{\perp,\max}^2 + \bar{x} M^2}{\bar{x} M^2} \right) - \frac{p_{\perp,\max}^2}{p_{\perp,\max}^2 + \bar{x} M^2} \right]$$
$$(10.46)$$

$$F_0(x) = \frac{v^2 + a^2}{8\pi^2} \frac{2\bar{x}}{x} \frac{p_{\perp,\max}^2}{p_{\perp,\max}^2 + \bar{x} M^2}$$
$$(10.47)$$

where $p_{\perp,\max}$ is the cutoff in transversal momentum, M is the vector-boson mass, v and a are the vector and axial-vector couplings, and $\bar{x} \equiv 1-x$. Note that the longitudinal structure function is finite for large cutoff, while the transversal structure function is logarithmically divergent.

The maximal transverse momentum is given by the kinematical limit, it is

$$p_{\perp,\max} = \bar{x}\sqrt{s}/2. \tag{10.48}$$

The vector and axial couplings for a fermion branching into a W are

$$v_W = \frac{g}{2\sqrt{2}}, \qquad\qquad a_W = \frac{g}{2\sqrt{2}}. \tag{10.49}$$

For Z emission, this is replaced by

$$v_Z = \frac{g}{2\cos\theta_w}\left(t_3 - 2q\sin^2\theta_w\right), \qquad a_Z = \frac{g}{2\cos\theta_w}t_3, \tag{10.50}$$

where $t_3 = \pm\frac{1}{2}$ is the fermion isospin, and q its charge.

For an initial antifermion, the signs of the axial couplings are inverted. Note that a common sign change of v and a is irrelevant.

The EWA depends on the parameters g, $\sin^2\theta_w$, M_W, and M_Z. These can all be taken from the SM input, and the prefactors are calculated from those and the incoming particle type.

Since these structure functions have a $1/x$ singularity (which is not really relevant in practice, however, since the vector boson mass is finite), we map this singularity allowing for nontrivial x bounds:

$$x = \exp(\bar{r}\ln x_0 + r\ln x_1) \tag{10.51}$$

such that

$$\int_{x_0}^{x_1} \frac{dx}{x} = (\ln x_1 - \ln x_0)\int_0^1 dr. \tag{10.52}$$

As a user parameter, we have the cutoff $p_{\perp,\max}$. The divergence $1/x$ also requires a x_0 cutoff; and for completeness we introduce a corresponding x_1. Physically, the minimal sensible value of x is M^2/s, although the approximation loses its value already at higher x values.

10.4.2 The EWA data block

The EWA parameters are: $p_{T,\max}$, c_V, c_A, and m. Instead of m we can use the incoming particle PDG code as input; from this we can deduce the mass and charges. In the initialization phase it is not yet determined whether a W or a Z is radiated, hence we set the vector and axial-vector couplings equal to the common prefactors $g/2 = e/2/\sin\theta_W$.

In principle, for EWA it would make sense top allow the user to also set the upper bound for x, $x_{\max}$, but we fix it to one here.

⟨*EWA: public*⟩≡
```
    public :: ewa_data_t
```

⟨*EWA: types*⟩≡
```
type :: ewa_data_t
   private
   type(model_t), pointer :: model => null ()
   type(flavor_t) :: flv
   real(default) :: pt_max
   real(default) :: sqrts
   real(default) :: x_min
   real(default) :: x_max
   real(default) :: mass
   real(default) :: q_min
   real(default) :: cv
   real(default) :: ca
   real(default) :: costhw
   real(default) :: sinthw
   real(default) :: mW
   real(default) :: mZ
   real(default) :: coeff
   logical(default) :: keep_momentum
   logical(default) :: keep_energy
   integer :: id = 0
   integer :: error = 0
end type ewa_data_t
```

Error codes

⟨*EWA: parameters*⟩≡
```
integer, parameter :: NONE = 0
integer, parameter :: ZERO_QMIN = 1
integer, parameter :: Q_MAX_TOO_SMALL = 2
integer, parameter :: ZERO_XMIN = 3
```

⟨*EWA: public*⟩+≡
```
public :: ewa_data_init
```

⟨*EWA: procedures*⟩≡
```
subroutine ewa_data_init (data, model, flv, x_min, q_min, pt_max, &
     sqrts, keep_momentum, keep_energy, mass)
  type(ewa_data_t), intent(inout) :: data
  type(model_t), intent(in), target :: model
  type(flavor_t), intent(in) :: flv
  real(default), intent(in) :: x_min, q_min, pt_max, sqrts
  logical, intent(in) :: keep_momentum, keep_energy
  real(default), intent(in), optional :: mass
  real(default) :: g, ee, sinthw
  data%model => model
  data%flv = flv
  data%pt_max = pt_max
  data%sqrts = sqrts
  data%x_min = x_min
  data%x_max = 1
  if (data%x_min == 0) then
     data%error = ZERO_XMIN;  return
  end if
  select case (char (model_get_name (data%model)))
  case ("QCD","QED","Test")
```

793

```
      call msg_fatal ("EWA structure function not available for model " &
           // char (model_get_name (data%model)) // ".")
   end select
   ee = model_get_parameter_value (data%model, var_str ("ee"))
   data%sinthw = model_get_parameter_value (data%model, var_str ("sw"))
   data%costhw = model_get_parameter_value (data%model, var_str ("cw"))
   data%mZ = model_get_parameter_value (data%model, var_str ("mZ"))
   data%mW = model_get_parameter_value (data%model, var_str ("mW"))
   if (data%sinthw /= 0) then
      g = ee / data%sinthw
   else
      call msg_fatal ("Vanishing value of sin(theta_w).")
   end if
   data%cv = g / 2._default
   data%ca = g / 2._default
   data%coeff = 1._default / (8._default * PI**2)
   data%keep_momentum = keep_momentum
   data%keep_energy = keep_energy
   if (present (mass)) then
      data%mass = mass
   else
      data%mass = flavor_get_mass (flv)
   end if
end subroutine ewa_data_init
```

Handle error conditions. Should always be done after initialization, unless we are sure everything is ok.

⟨*EWA: public*⟩+≡
```
    public :: ewa_data_check
```

⟨*EWA: procedures*⟩+≡
```
    subroutine ewa_data_check (data)
      type(ewa_data_t), intent(in) :: data
      select case (data%error)
      case (ZERO_QMIN)
         call msg_fatal (" EWA: Particle mass is zero")
      case (Q_MAX_TOO_SMALL)
         call msg_fatal (" EWA: Particle mass exceeds Qmax")
      case (ZERO_XMIN)
         call msg_fatal (" EWA: x_min must be larger than zero")
      end select
    end subroutine ewa_data_check
```

Set the vector boson ID for distinguishing W and Z bosons.

⟨*EWA: public*⟩+≡
```
    public :: ewa_set_id
```

⟨*EWA: procedures*⟩+≡
```
    subroutine ewa_set_id (data, id)
      type (ewa_data_t), intent(inout) :: data
      integer, intent(in) :: id
      data%id = id
    end subroutine ewa_set_id
```

Output

⟨*EWA: public*⟩+≡
```
    public :: ewa_data_write
```

⟨*EWA: procedures*⟩+≡
```
    subroutine ewa_data_write (data, unit)
      type(ewa_data_t), intent(in) :: data
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, *) "EWA data:"
      write (u, *) "  prt      = ", char (flavor_get_name (data%flv))
      write (u, *) "  x_min    = ", data%x_min
      write (u, *) "  x_max    = ", data%x_max
      write (u, *) "  pt_max   = ", data%pt_max
      write (u, *) "  sqrts    = ", data%sqrts
      write (u, *) "  mass     = ", data%mass
      write (u, *) "  cv       = ", data%cv
      write (u, *) "  ca       = ", data%ca
      write (u, *) "  coeff    = ", data%coeff
      write (u, *) "  costhw   = ", data%costhw
      write (u, *) "  sinthw   = ", data%sinthw
      write (u, *) "  mZ       = ", data%mZ
      write (u, *) "  mW       = ", data%mW
      write (u, *) "  keep_mom. = ", data%keep_momentum
      write (u, *) "  keep_en.  = ", data%keep_energy
    end subroutine ewa_data_write
```

10.4.3 The EWA object

The `ewa_t` data type is a $1 \to 2$ interaction. We should be able to handle several flavors in parallel, since EWA is not necessarily applied immediately after beam collision: Photons may be radiated from quarks. In that case, the partons are massless and $q_{\min}$ applies instead, so we do not need to generate several kinematical configurations in parallel.

The particles are ordered as (incoming, radiated, W/Z), where the W/Z initiates the hard interaction.

For EPA, we generate an unpolarized photon and transfer initial polarization to the radiated parton. Color is transferred in the same way. I do not know whether the same can/should be done for EWA, as the structure functions depend on the W/Z polarization. If we are having Z bosons, both up- and down-type fermions can participate. Otherwise, with a W^+ an up-type fermion is transferred to a down-type fermion, and the other way round.

⟨*EWA: public*⟩+≡
```
    public :: interaction_init_ewa
```

⟨*EWA: procedures*⟩+≡
```
    subroutine interaction_init_ewa (int, data)
      type(interaction_t), intent(out) :: int
      type(ewa_data_t), intent(in) :: data
      type(quantum_numbers_mask_t), dimension(3) :: mask
      integer, dimension(3) :: lock
```

```
type(polarization_t) :: pol
type(quantum_numbers_t), dimension(1) :: qn_fc, qn_hel, qn_fc_fin
type(flavor_t) :: flv_z, flv_wp, flv_wm, flv_down, flv_up
type(quantum_numbers_t) :: qn_z, qn_wp, qn_wm, qn, qn_down, &
     qn_up
type(state_iterator_t) :: it_hel
real(default) :: q, t3
integer :: i, isospin, pdg
logical :: up_type
isospin = flavor_get_isospin_type (data%flv)
pdg = flavor_get_pdg (data%flv)
if (abs(isospin) /= 2 .or. abs(pdg) > 16 .or. pdg == 0 .or. pdg == 9) then
   call msg_fatal ("EWA structure function only accessible for " &
        // "SM quarks and leptons.")
end if
q = - flavor_get_charge (data%flv)
t3 = - flavor_get_isospin (data%flv)
up_type = (t3 > 0)
mask = new_quantum_numbers_mask (.false., .false., &
     mask_h = (/ .false., .false., .true. /))
lock = (/ 2, 1, 0 /)
call interaction_init &
     (int, 1, 0, 2, mask=mask, lock=lock, set_relations=.true.)
select case (data%id)
case (23)
   !!! Z boson, flavor is not changing
   call flavor_init (flv_z, Z_BOSON, data%model)
   call quantum_numbers_init (qn_z, flv_z)
   call polarization_init_generic (pol, data%flv)
   call quantum_numbers_init (qn_fc(1), &
           flv = data%flv, col = color_from_flavor (data%flv))
   call state_iterator_init (it_hel, pol%state)
   do while (state_iterator_is_valid (it_hel))
      qn_hel = state_iterator_get_quantum_numbers (it_hel)
      qn = qn_hel(1) .merge. qn_fc(1)
      call interaction_add_state (int, (/ qn, qn, qn_z /))
      call state_iterator_advance (it_hel)
   end do
   call polarization_final (pol)
case (24)
   if (up_type) then
      !!! W+, flavor changing
      call flavor_init (flv_down, pdg - 1, data%model)
      call flavor_init (flv_wp, W_BOSON, data%model)
      call quantum_numbers_init (qn_wp, flv_wp)
      call polarization_init_generic (pol, data%flv)
      call quantum_numbers_init (qn_fc(1), &
              flv = data%flv, col = color_from_flavor (data%flv))
      call quantum_numbers_init (qn_fc_fin(1), &
             flv = flv_down, col = color_from_flavor (flv_down))
      call state_iterator_init (it_hel, pol%state)
      do while (state_iterator_is_valid (it_hel))
         qn_hel = state_iterator_get_quantum_numbers (it_hel)
         qn = qn_hel(1) .merge. qn_fc(1)
```

```
                qn_down = qn_hel(1) .merge. qn_fc_fin(1)
                call interaction_add_state (int, (/ qn, qn_down, qn_wp /))
                call state_iterator_advance (it_hel)
             end do
             call polarization_final (pol)
          else
             !!! W-, flavor changing
             call flavor_init (flv_up, pdg + 1, data%model)
             call flavor_init (flv_wm, - W_BOSON, data%model)
             call quantum_numbers_init (qn_wm, flv_wm)
             call polarization_init_generic (pol, data%flv)
             call quantum_numbers_init (qn_fc(1), &
                   flv = data%flv, col = color_from_flavor (data%flv))
             call quantum_numbers_init (qn_fc_fin(1), &
                   flv = flv_up, col = color_from_flavor (flv_up))
             call state_iterator_init (it_hel, pol%state)
             do while (state_iterator_is_valid (it_hel))
                qn_hel = state_iterator_get_quantum_numbers (it_hel)
                qn = qn_hel(1) .merge. qn_fc(1)
                qn_up = qn_hel(1) .merge. qn_fc_fin(1)
                call interaction_add_state (int, (/ qn, qn_up, qn_wm /))
                call state_iterator_advance (it_hel)
             end do
             call polarization_final (pol)
          end if
       case default
          call msg_fatal ("EWA initialization failed: wrong particle type.")
       end select
       call interaction_freeze (int)
     end subroutine interaction_init_ewa
```

10.4.4 EWA structure function

The EWA structure function allows for a straightforward mapping of the unit
interval. So, to leading order, the structure function value is unity, but the x
value is transformed. Higher orders affect the function value.

The structure function implementation applies the above mapping to the
input (random) number r to generate the momentum fraction x and the function
value f. For numerical stability reasons, we also output xb, which is $\bar{x} =
1 - x$. The mapping ensures that $f = 1$ at leading order, and the higher-order
corrections are well-behaved.

⟨*EWA: procedures*⟩+≡
```
   elemental subroutine strfun (f, fp, fm, fL, x, xb, r, E, data)
     real(default), intent(out) :: f, fm, fp, fL
     real(default) :: fsum
     real(default), intent(out) :: x, xb
     real(default), intent(in) :: r, E
     type(ewa_data_t), intent(in) :: data
     real(default) :: x0, x1
     real(default) :: rb, lx0, lx1, lx, d, den
     real(default) :: c1, c2, pt2
     real(default) :: costhw, sinthw
```

```fortran
real(default) :: cv, ca, q, t3
if (data%keep_momentum .or. data%keep_energy) then
   select case (data%id)
   case (23)
      x0 = max (data%x_min, data%mz/E)
   case (24)
      x0 = max (data%x_min, data%mw/E)
   end select
else
   x0 = data%x_min
end if
x1 = data%x_max
if (x0 >= x1) then
   f = 0
   fp = 0
   fm = 0
   fL = 0
   return
end if
lx0 = log (x0)
lx1 = log (x1)
lx = lx1 * r + lx0 * (1-r)
x = exp(lx)
xb = 1 - x
f = data%coeff * (lx1 - lx0)
pt2 = min ((data%pt_max)**2, (xb * data%sqrts / 2)**2)
select case (data%id)
case (23)
   !!! Z boson structure function
   c1 = log (1 + pt2 / (xb * (data%mZ)**2))
   c2 = 1 / (1 + (xb * (data%mZ)**2) / pt2)
   if (flavor_is_antiparticle (data%flv)) then
      q = flavor_get_charge (data%flv)
      t3 = flavor_get_isospin (data%flv)
   else
      q = - flavor_get_charge (data%flv)
      t3 = - flavor_get_isospin (data%flv)
   end if
   cv = data%cv * (t3 - 2._default * q * data%sinthw**2) / data%costhw
   ca = data%ca *  t3 / data%costhw
   if (flavor_is_antiparticle (data%flv)) ca = - ca
   fm = ((cv + ca)**2 + ((cv - ca) * xb)**2) / 2 * (c1 - c2)
   fp = ((cv - ca)**2 + ((cv + ca) * xb)**2) / 2 * (c1 - c2)
   fL = (cv**2 + ca**2) * 2 * xb * c2
   fsum = fp + fm + fL
   f = f * fsum
   if (fsum /= 0) then
      fp = fp / fsum
      fm = fm / fsum
      fL = fL / fsum
   end if
case (24)
   !!! W boson structure function
   c1 = log (1 + pt2 / (xb * (data%mW)**2))
```

```
        c2 = 1 / (1 + (xb * (data%mW)**2) / pt2)
        cv = data%cv / sqrt(2._default)
        ca = data%ca / sqrt(2._default)
        if (flavor_is_antiparticle (data%flv)) ca = - ca
        fm = ((cv + ca)**2 + ((cv - ca) * xb)**2) / 2 * (c1 - c2)
        fp = ((cv - ca)**2 + ((cv + ca) * xb)**2) / 2 * (c1 - c2)
        fL = (cv**2 + ca**2) * 2 * xb * c2
        fsum = fp + fm + fL
        f = f * fsum
        if (fsum /= 0) then
           fp = fp / fsum
           fm = fm / fsum
           fL = fL / fsum
        end if
     end select
   end subroutine strfun
```

10.4.5   EWA application

For EWA, we can compute kinematics and function value in a single step. This
function works on a single beam, assuming that the input momentum has been
set. We need four random numbers as input: one for x, one for Q^2, and two
for the polar and azimuthal angles. Alternatively, we can skip p_T generation;
in this case, we only need one.

   For obtaining splitting kinematics, we rely on the assumption that all in-
particles are mass-degenerate (or there is only one), so the generated x values
are identical.

⟨*EWA: public*⟩+≡
```
    public :: interaction_apply_ewa
```

⟨*EWA: procedures*⟩+≡
```
    subroutine interaction_apply_ewa (int, r, ewa_data)
      type(interaction_t), intent(inout) :: int
      real(default), dimension(:), intent(in) :: r
      type(ewa_data_t), dimension(:), intent(in) :: ewa_data
      type(vector4_t) :: k
      type(vector4_t), dimension(2) :: k_split
      type(splitting_data_t) :: sd
      real(default), dimension(size(ewa_data)) :: f, fp, fm, fL
      real(default), dimension(size(ewa_data)) :: x, xb
      k = interaction_get_momentum (int, 1)
      select case (ewa_data(1)%id)
      case (23)
         sd = new_splitting_data (k, k**2, ewa_data(1)%mass**2, ewa_data(1)%mz)
      case (24)
         sd = new_splitting_data (k, k**2, ewa_data(1)%mass**2, ewa_data(1)%mw)
      end select
      call strfun (f, fp, fm, fL, x, xb, r(1), energy (k), ewa_data(1))
      call interaction_set_flavored_values &
           (int, cmplx (f, kind=default), ewa_data%flv, 2)
      call splitting_set_t_bounds (sd, x(1), xb(1))
      call splitting_narrow_t_bounds (sd, ewa_data(1)%q_min, ewa_data(1)%pt_max)
```

```
select case (size (r))
 case (1)
    call splitting_set_collinear (sd)
 case (3)
    call splitting_sample_t (sd, r(2))
    call splitting_sample_phi (sd, r(3))
 case default
    print *, "n_rand = ", size (r)
    call msg_bug (" EWA: number of random numbers must be 1 or 3")
 end select
 k_split = split_momentum (k, sd)
 if (ewa_data(1)%keep_momentum) then
    call on_shell (k_split(1), 0._default, KEEP_MOMENTUM)
    select case (ewa_data(1)%id)
    case (23)
       call on_shell (k_split(2), ewa_data(1)%mz, KEEP_MOMENTUM)
    case (24)
       call on_shell (k_split(2), ewa_data(1)%mw, KEEP_MOMENTUM)
    end select
 else if (ewa_data(1)%keep_energy) then
    call on_shell (k_split(1), 0._default, KEEP_ENERGY)
    select case (ewa_data(1)%id)
    case (23)
       call on_shell (k_split(2), ewa_data(1)%mz, KEEP_ENERGY)
    case (24)
       call on_shell (k_split(2), ewa_data(1)%mw, KEEP_ENERGY)
    end select
 end if
 call interaction_set_momenta &
      (int, k_split, outgoing=.true.)
end subroutine interaction_apply_ewa
```

10.5 LHAPDF

Parton distribution functions (PDFs) are available via an interface to the LHAPDF standard library.

The default PDF for protons set is chosen to be CTEQ6ll (LO fit with LO α_s).

⟨*Limits: public parameters*⟩+≡
```
    character(*), parameter, public :: LHAPDF_DEFAULT_PROTON = "cteq6ll.LHpdf"
    character(*), parameter, public :: LHAPDF_DEFAULT_PION   = "ABFKWPI.LHgrid"
    character(*), parameter, public :: LHAPDF_DEFAULT_PHOTON = "GSG960.LHgrid"
```

10.5.1 The module

⟨`sf_lhapdf.f90`⟩≡
  ⟨*File header*⟩

  ```
  module sf_lhapdf
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
```
    use system_dependencies, only: LHAPDF_PDFSETS_PATH !NODEP!
    use system_dependencies, only: LHAPDF_AVAILABLE !NODEP!
    use limits, only: LHAPDF_DEFAULT_PROTON !NODEP!
    use limits, only: LHAPDF_DEFAULT_PION !NODEP!
    use limits, only: LHAPDF_DEFAULT_PHOTON !NODEP!
```
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use models
    use flavors
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use sf_aux
```

  ⟨*Standard module head*⟩

  ⟨*LHAPDF: public*⟩

  ⟨*LHAPDF: types*⟩

  ⟨*LHAPDF: interfaces*⟩

  ```
  contains
```

  ⟨*LHAPDF: procedures*⟩

  ```
  end module sf_lhapdf
```

10.5.2 LHAPDF library interface

Here we specify explicit interfaces for all LHAPDF routines that we use below.

⟨*LHAPDF: interfaces*⟩≡
```
interface
   subroutine InitPDFsetM (set, file)
     integer, intent(in) :: set
     character(*), intent(in) :: file
   end subroutine InitPDFsetM
end interface
```

⟨*LHAPDF: interfaces*⟩+≡
```
interface
   subroutine InitPDFM (set, mem)
     integer, intent(in) :: set, mem
   end subroutine InitPDFM
end interface
```

⟨*LHAPDF: interfaces*⟩+≡
```
interface
   subroutine numberPDFM (set, n_members)
     integer, intent(in) :: set
     integer, intent(out) :: n_members
   end subroutine numberPDFM
end interface
```

⟨*LHAPDF: interfaces*⟩+≡
```
interface
   subroutine evolvePDFM (set, x, q, ff)
     integer, intent(in) :: set
     double precision, intent(in) :: x, q
     double precision, dimension(-6:6), intent(out) :: ff
   end subroutine evolvePDFM
end interface
```

⟨*LHAPDF: interfaces*⟩+≡
```
interface
   subroutine evolvePDFpM (set, x, q, s, scheme, ff)
     integer, intent(in) :: set
     double precision, intent(in) :: x, q, s
     integer, intent(in) :: scheme
     double precision, dimension(-6:6), intent(out) :: ff
   end subroutine evolvePDFpM
end interface
```

⟨*LHAPDF: interfaces*⟩+≡
```
interface
   subroutine GetXminM (set, mem, xmin)
     integer, intent(in) :: set, mem
     double precision, intent(out) :: xmin
   end subroutine GetXminM
end interface
```

⟨*LHAPDF: interfaces*⟩+≡
```
  interface
     subroutine GetXmaxM (set, mem, xmax)
       integer, intent(in) :: set, mem
       double precision, intent(out) :: xmax
     end subroutine GetXmaxM
  end interface
```

⟨*LHAPDF: interfaces*⟩+≡
```
  interface
     subroutine GetQ2minM (set, mem, q2min)
       integer, intent(in) :: set, mem
       double precision, intent(out) :: q2min
     end subroutine GetQ2minM
  end interface
```

⟨*LHAPDF: interfaces*⟩+≡
```
  interface
     subroutine GetQ2maxM (set, mem, q2max)
       integer, intent(in) :: set, mem
       double precision, intent(out) :: q2max
     end subroutine GetQ2maxM
  end interface
```

10.5.3 The LHAPDF status

This type holds the initialization status of the LHAPDF system.

⟨*LHAPDF: public*⟩≡
```
    public :: lhapdf_status_t
```

⟨*LHAPDF: types*⟩≡
```
    type :: lhapdf_status_t
       private
       logical, dimension(3) :: initialized = .false.
    end type lhapdf_status_t
```

⟨*LHAPDF: public*⟩+≡
```
    public :: lhapdf_status_reset
```

⟨*LHAPDF: procedures*⟩≡
```
    subroutine lhapdf_status_reset (lhapdf_status)
      type(lhapdf_status_t), intent(inout) :: lhapdf_status
      lhapdf_status%initialized = .false.
    end subroutine lhapdf_status_reset
```

⟨*LHAPDF: procedures*⟩+≡
```
    function lhapdf_status_is_initialized (lhapdf_status, set) result (flag)
      logical :: flag
      type(lhapdf_status_t), intent(in) :: lhapdf_status
      integer, intent(in), optional :: set
      if (present (set)) then
         select case (set)
```

```
          case (1:3);    flag = lhapdf_status%initialized(set)
          case default;  flag = .false.
          end select
       else
          flag = any (lhapdf_status%initialized)
       end if
    end function lhapdf_status_is_initialized
```

⟨*LHAPDF: procedures*⟩+≡
```
    subroutine lhapdf_status_set_initialized (lhapdf_status, set)
      type(lhapdf_status_t), intent(inout) :: lhapdf_status
      integer, intent(in) :: set
      lhapdf_status%initialized(set) = .true.
    end subroutine lhapdf_status_set_initialized
```

10.5.4   LHAPDF initialization

Before using LHAPDF, we have to initialize it with a particular data set and
member. This applies not just if we use structure functions, but also if we just
use an α_s formula. The integer set should be 1 for proton, 2 for pion, and 3
for photon, but this is just convention.

   If the particular set has already been initialized, do nothing. This implies
that whenever we want to change the setup for a particular set, we have to reset
the LHAPDF status.

⟨*LHAPDF: public*⟩+≡
```
    public :: lhapdf_init
```

⟨*LHAPDF: procedures*⟩+≡
```
    subroutine lhapdf_init (status, set, prefix, file, member)
      type(lhapdf_status_t), intent(inout) :: status
      integer, intent(in) :: set
      type(string_t), intent(in) :: prefix
      type(string_t), intent(inout) :: file
      integer, intent(inout) :: member
      if (lhapdf_status_is_initialized (status, set))  return
      if (file == "") then
         select case (set)
         case (1);  file = LHAPDF_DEFAULT_PROTON
         case (2);  file = LHAPDF_DEFAULT_PION
         case (3);  file = LHAPDF_DEFAULT_PHOTON
         end select
      end if
      if (data_file_exists (prefix // file)) then
         call InitPDFsetM (set, char (prefix // file))
      else
         call msg_fatal ("LHAPDF: Data file '" // char (file) // "' not found.")
         return
      end if
      if (.not. dataset_member_exists (set, member)) then
         call msg_error (" LHAPDF: Chosen member does not exist for set '" &
               // char (file) // "', using default.")
         member = 0
```

804

```
      end if
      call InitPDFM (set, member)
      call lhapdf_status_set_initialized (status, set)
    contains
      function data_file_exists (fq_name) result (exist)
        type(string_t), intent(in) :: fq_name
        logical :: exist
        inquire (file = char(fq_name), exist = exist)
      end function data_file_exists
      function dataset_member_exists (set, member) result (exist)
        integer, intent(in) :: set, member
        logical :: exist
        integer :: n_members
        call numberPDFM (set, n_members)
        exist = member >= 0 .and. member <= n_members
      end function dataset_member_exists
    end subroutine lhapdf_init
```

10.5.5 The LHAPDF data block

The data block holds the incoming flavor (which has to be proton, pion, or
photon), the corresponding pointer to the global access data (1, 2, or 3), the
flag `invert` which is set for an antiproton, the bounds as returned by LHAPDF
for the specified set, and a mask that determines which partons will be actually
in use.

⟨*LHAPDF: public*⟩+≡
```
    public :: lhapdf_data_t
```

⟨*LHAPDF: types*⟩+≡
```
    type :: lhapdf_data_t
      private
      type(string_t) :: prefix
      type(string_t) :: file
      integer :: member = 0
      type(model_t), pointer :: model => null ()
      type(flavor_t) :: flv_in
      integer :: set = 0
      logical :: invert = .false.
      logical :: photon = .false.
      integer :: photon_scheme = 0
      real(default) :: xmin = 0, xmax = 0
      real(default) :: qmin = 0, qmax = 0
      logical, dimension(-6:6) :: mask = .true.
    end type lhapdf_data_t
```

Generate PDF data. This is a provided as a function, but it has the side-effect
of initializing the requested PDF set. A finalizer is not needed.

The library uses double precision, so since the default precision may be
quadruple, we use auxiliary variables for type casting.

⟨*LHAPDF: public*⟩+≡
```
    public :: lhapdf_data_init
```

⟨*LHAPDF: procedures*⟩+≡

```
    subroutine lhapdf_data_init &
        (data, status, model, flv, file, member, photon_scheme)
      type(lhapdf_data_t), intent(out) :: data
      type(lhapdf_status_t), intent(inout) :: status
      type(model_t), intent(in), target :: model
      type(flavor_t), intent(in) :: flv
      type(string_t), intent(in), optional :: file
      integer, intent(in), optional :: member
      integer, intent(in), optional :: photon_scheme
      integer :: mem
      double precision :: xmin, xmax, q2min, q2max
      external :: InitPDFsetM, InitPDFM, numberPDFM
      external :: GetXminM, GetXmaxM, GetQ2minM, GetQ2maxM
      if (.not. LHAPDF_AVAILABLE) then
         call msg_fatal ("LHAPDF requested but library is not linked")
         return
      end if
      data%model => model
      data%flv_in = flv
      select case (flavor_get_pdg (flv))
      case (PROTON)
         data%set = 1
      case (-PROTON)
         data%set = 1
         data%invert = .true.
      case (PIPLUS)
         data%set = 2
      case (-PIPLUS)
         data%set = 2
         data%invert = .true.
      case (PHOTON)
         data%set = 3
         data%photon = .true.
         if (present (photon_scheme))  data%photon_scheme = photon_scheme
      case default
         call msg_fatal (" LHAPDF: " &
             // "incoming particle must be (anti)proton, pion, or photon.")
         return
      end select
      data%prefix = LHAPDF_PDFSETS_PATH // "/"
      if (present (file)) then
         data%file = file
      else
         data%file = ""
      end if
      call lhapdf_init (status, data%set, data%prefix, data%file, data%member)
      call GetXminM (data%set, data%member, xmin)
      call GetXmaxM (data%set, data%member, xmax)
      call GetQ2minM (data%set, data%member, q2min)
      call GetQ2maxM (data%set, data%member, q2max)
      data%xmin = xmin
      data%xmax = xmax
      data%qmin = sqrt (q2min)
```

```
        data%qmax = sqrt (q2max)
    end subroutine lhapdf_data_init
```

Enable/disable partons explicitly. If a mask entry is true, applying the PDF
will generate the corresponding flavor on output.

⟨*LHAPDF: public*⟩+≡
```
    public :: lhapdf_data_set_mask
```

⟨*LHAPDF: procedures*⟩+≡
```
    subroutine lhapdf_data_set_mask (data, mask)
      type(lhapdf_data_t), intent(inout) :: data
      logical, dimension(-6:6), intent(in) :: mask
      data%mask = mask
    end subroutine lhapdf_data_set_mask
```

Output

⟨*LHAPDF: public*⟩+≡
```
    public :: lhapdf_data_write
```

⟨*LHAPDF: procedures*⟩+≡
```
    subroutine lhapdf_data_write (data, unit)
      type(lhapdf_data_t), intent(in) :: data
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, *) "LHAPDF data:"
      if (data%set /= 0) then
         write (u, "(3x,A)", advance="no") "flavor       = "
         call flavor_write (data%flv_in, u);  write (u, *)
         write (u, *) "  prefix      = ", char (data%prefix)
         write (u, *) "  file        = ", char (data%file)
         write (u, *) "  member      = ", data%member
         write (u, *) "  x(min)      = ", data%xmin
         write (u, *) "  x(max)      = ", data%xmax
         write (u, *) "  Q^2(min)    = ", data%qmin
         write (u, *) "  Q^2(max)    = ", data%qmax
         write (u, *) "  invert      = ", data%invert
         if (data%photon)  write (u, *) "  IP2 (scheme) = ", data%photon_scheme
         write (u, *) "  mask        = ", &
              data%mask(-6:-1), "*", data%mask(0), "*", data%mask(1:6)
      else
         write (u, *) "  [undefined]"
      end if
    end subroutine lhapdf_data_write
```

10.5.6   The LHAPDF object

The `lhapdf_t` data type is a $1 \rightarrow 2$ interaction which describes the splitting
of an (anti)proton into a parton and a beam remnant. We stay in the strict
forward-splitting limit, but allow some invariant mass for the beam remnant
such that the outgoing parton is exactly massless. For a real event, we would

replace this by a parton cascade, where the outgoing partons have virtuality as dictated by parton-shower kinematics, and transverse momentum is generated.

This is the LHAPDF object which holds input data together with the interaction. We also store the x momentum fraction and the scale, since kinematics and function value are requested at different times.

The PDF application is a $1 \rightarrow 2$ splitting process, where the particles are ordered as (hadron, remnant, parton).

Polarization is ignored completely. The beam particle is colorless, while partons and beam remnant carry color. The remnant gets a special flavor code.

⟨*LHAPDF: public*⟩+≡

```
public :: interaction_init_lhapdf
```

⟨*LHAPDF: procedures*⟩+≡

```
subroutine interaction_init_lhapdf (int, data)
  type(interaction_t), intent(out) :: int
  type(lhapdf_data_t), intent(in) :: data
  type(quantum_numbers_mask_t), dimension(3) :: mask
  type(quantum_numbers_t) :: qn_beam, qn_remnant, qn_parton
  type(flavor_t) :: flv, flv_remnant
  integer :: i
  mask = new_quantum_numbers_mask (.false., .false., .true.)
  call interaction_init (int, 1, 0, 2, mask=mask, set_relations=.true.)
  call quantum_numbers_init (qn_beam, flv = data%flv_in)
  do i = -6, 6
     if (data%mask(i)) then
        if (i == 0) then
           call flavor_init (flv, GLUON, data%model)
           call flavor_init (flv_remnant, HADRON_REMNANT_OCTET, data%model)
        else
           call flavor_init (flv, i, data%model)
           call flavor_init (flv_remnant, &
                 sign (HADRON_REMNANT_TRIPLET, -i), data%model)
        end if
        call quantum_numbers_init (qn_remnant, &
              flv = flv_remnant, col = color_from_flavor (flv_remnant, 1))
        call quantum_numbers_init (qn_parton, &
              flv = flv, col = color_from_flavor (flv, 1, reverse=.true.))
        call interaction_add_state (int, &
              (/ qn_beam, qn_remnant, qn_parton /))
     end if
  end do
  call interaction_freeze (int)
end subroutine interaction_init_lhapdf
```

10.5.7 Structure function

For the PDFs, we separate kinematics from dynamics. We first generate appropriate x values, independent of flavor and scale. This allows us to determine the momenta that initiate the hard scattering. Only when the hard process has been computed, we can be sure about the scattering scale and compute the structure function values.

For the x values, we can only apply a simple mapping that eliminates the $1/x$ singularity, i.e., we generate x on a logarithmic scale. This produces a Jacobian factor that we store along with the generated x value. The boundaries are used as returned by the LHAPDF library for the chosen PDF set.

⟨*LHAPDF: procedures*⟩+≡
```
    subroutine generate_x (x, f, r, lhapdf_data)
      real(default), intent(out) :: x, f
      real(default), intent(in) :: r
      type(lhapdf_data_t), intent(in) :: lhapdf_data
      real(default) :: lg
      x = r
      f = 1
!     lg = log (lhapdf_data% xmax / lhapdf_data% xmin)
!     x = lhapdf_data% xmin * exp (r * lg)
!     f = x * lg
    end subroutine generate_x
```

The previous routine is called here, so we can compute the complete kinematics:

⟨*LHAPDF: public*⟩+≡
```
    public :: interaction_set_kinematics_lhapdf
```

⟨*LHAPDF: procedures*⟩+≡
```
    subroutine interaction_set_kinematics_lhapdf (int, x, f, s, r, lhapdf_data)
      type(interaction_t), intent(inout) :: int
      real(default), intent(out) :: x, f, s
      real(default), intent(in) :: r
      type(lhapdf_data_t), intent(in) :: lhapdf_data
      type(vector4_t) :: k
      type(splitting_data_t) :: sd
      call generate_x (x, f, r, lhapdf_data)
      k = interaction_get_momentum (int, 1)
      s = k**2
      sd = new_splitting_data (k,  s, 0._default, 0._default)
      call splitting_set_t_bounds (sd,  x, 1 -  x)
      call splitting_set_collinear (sd)
      call interaction_set_momenta &
           (int, split_momentum (k, sd), outgoing=.true.)
    end subroutine interaction_set_kinematics_lhapdf
```

Once the scale is also known, we can actually call the library and set the values. If the scale is out of bounds, we reset it to the boundary. Account for the Jacobian.

We have to cast the LHAPDF arguments to/from double precision (possibly from/to quadruple precision), if necessary.

⟨*LHAPDF: public*⟩+≡
```
    public :: interaction_apply_lhapdf
```

⟨*LHAPDF: procedures*⟩+≡
```
    subroutine interaction_apply_lhapdf (int, scale, x, f, s, lhapdf_data)
      type(interaction_t), intent(inout) :: int
      real(default), intent(in) :: scale, x, f, s
      type(lhapdf_data_t), intent(in) :: lhapdf_data
      double precision :: xx, qq, ss
```

```
      double precision, dimension(-6:6) :: ff
      complex(default), dimension(:), allocatable :: fc
      external :: evolvePDFM, evolvePDFpM
      xx = x
      qq = min (lhapdf_data% qmax, scale)
      qq = max (lhapdf_data% qmin, qq)
      if (.not. lhapdf_data% photon) then
         if (lhapdf_data% invert) then
            call evolvePDFM (lhapdf_data% set, xx, qq, ff(6:-6:-1))
         else
            call evolvePDFM (lhapdf_data% set, xx, qq, ff)
         end if
      else
         ss = s
         call evolvePDFpM (lhapdf_data% set, xx, qq, &
              ss, lhapdf_data% photon_scheme, ff)
      end if
      allocate (fc (count (lhapdf_data% mask)))
      fc = pack (ff / x, lhapdf_data% mask) * f
      call interaction_set_matrix_element (int, fc)
    end subroutine interaction_apply_lhapdf
```

10.6   Spectra and structure functions: wrapper

In this module, we collect, for each type of spectrum or structure function, the
data, initialization routines, and applications.

⟨strfun.f90⟩≡
  ⟨*File header*⟩

```
  module strfun
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use models
    use quantum_numbers
    use interactions
    use evaluators
    use beams
    use sf_isr
    use sf_epa
    use sf_ewa
    use sf_lhapdf
```

  ⟨*Standard module head*⟩

  ⟨*Strfun: public*⟩

  ⟨*Strfun: parameters*⟩

⟨*Strfun: types*⟩

⟨*Strfun: interfaces*⟩

```
contains
```

⟨*Strfun: procedures*⟩

```
end module strfun
```

10.6.1   The structure functions type

Definition

This contains the specific structure function data, much of which depends on the type. An extensible type would be appropriate. As long as this is not available in general, we emulate it by allocating the requested data explicitly.

⟨*Strfun: public*⟩≡
```
  !  public :: strfun_t
```

⟨*Strfun: types*⟩≡
```
   type :: strfun_t
     private
     integer :: type = STRF_NONE
     type(string_t) :: name
     type(interaction_t) :: int
     type(lhapdf_data_t), dimension(:), allocatable :: lhapdf_data
     type(isr_data_t), dimension(:), allocatable :: isr_data
     type(epa_data_t), dimension(:), allocatable :: epa_data
     type(ewa_data_t), dimension(:), allocatable :: ewa_data
     real(default) :: x = 0, f = 1, s = 0
     real(default) :: scale = 0
   end type strfun_t
```

The list of structure function codes:

⟨*Strfun: parameters*⟩≡
```
    integer, parameter, public :: STRF_NONE = 0
    integer, parameter, public :: STRF_LHAPDF = 1, STRF_ISR = 2, &
        STRF_EPA = 3, STRF_EWA = 4
```

The initializer assigns specific data and tags the interaction. The data block(s) have to be known already.

⟨*Strfun: public*⟩+≡
```
  !  public :: strfun_init
```

⟨*Strfun: interfaces*⟩≡
```
   interface strfun_init
     module procedure strfun_init_lhapdf
     module procedure strfun_init_isr
     module procedure strfun_init_epa
     module procedure strfun_init_ewa
   end interface
```

⟨*Strfun: procedures*⟩≡
```
    subroutine strfun_init_lhapdf (strfun, lhapdf_data)
      type(strfun_t), intent(out) :: strfun
      type(lhapdf_data_t), intent(in) :: lhapdf_data
      strfun%type = STRF_LHAPDF
      strfun%name = "LHAPDF"
      allocate (strfun%lhapdf_data (1))
      strfun%lhapdf_data = lhapdf_data
      call interaction_init_lhapdf (strfun%int, lhapdf_data)
    end subroutine strfun_init_lhapdf

    subroutine strfun_init_isr (strfun, isr_data)
      type(strfun_t), intent(out) :: strfun
      type(isr_data_t), intent(in) :: isr_data
      strfun%type = STRF_ISR
      strfun%name = "ISR"
      allocate (strfun%isr_data (1))
      strfun%isr_data = isr_data
      call interaction_init_isr (strfun%int, isr_data)
    end subroutine strfun_init_isr

    subroutine strfun_init_epa (strfun, epa_data)
      type(strfun_t), intent(out) :: strfun
      type(epa_data_t), intent(in) :: epa_data
      strfun%type = STRF_EPA
      strfun%name = "EPA"
      allocate (strfun%epa_data (1))
      strfun%epa_data = epa_data
      call interaction_init_epa (strfun%int, epa_data)
    end subroutine strfun_init_epa

    subroutine strfun_init_ewa (strfun, ewa_data, id)
      type(strfun_t), intent(out) :: strfun
      type(ewa_data_t), intent(inout) :: ewa_data
      integer, intent(in) :: id
      strfun%type = STRF_EWA
      strfun%name = "EWA"
      allocate (strfun%ewa_data (1))
      call ewa_set_id (ewa_data, id)
      strfun%ewa_data = ewa_data
      call interaction_init_ewa (strfun%int, ewa_data)
    end subroutine strfun_init_ewa
```

Finalizer for the contained interaction:

⟨*Strfun: public*⟩+≡
```
  !  public :: strfun_final
```

⟨*Strfun: procedures*⟩+≡
```
    elemental subroutine strfun_final (strfun)
      type(strfun_t), intent(inout) :: strfun
      select case (strfun%type)
      case (STRF_ISR)
         deallocate (strfun%isr_data)
      case (STRF_EPA)
```

```
            deallocate (strfun%epa_data)
         case (STRF_EWA)
            deallocate (strfun%ewa_data)
         case (STRF_LHAPDF)
            deallocate (strfun%lhapdf_data)
         end select
         call interaction_final (strfun%int)
         strfun%type = STRF_NONE
      end subroutine strfun_final
```

I/O

⟨*Strfun: public*⟩+≡
```
   !  public :: strfun_write
```

⟨*Strfun: procedures*⟩+≡
```
      subroutine strfun_write (strfun, unit, verbose, show_momentum_sum, show_mass)
         type(strfun_t), intent(in) :: strfun
         integer, intent(in), optional :: unit
         logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
         integer :: u
         u = output_unit (unit);  if (u < 0)  return
         if (strfun%type /= STRF_NONE) then
            write (u, *) char (strfun_get_name (strfun)) // " setup:"
            select case (strfun%type)
            case (STRF_LHAPDF)
               call lhapdf_data_write (strfun%lhapdf_data(1), u)
               write (u, *) "LHAPDF event data:"
               write (u, *) "  x     =", strfun%x
               write (u, *) "  f     =", strfun%f
               write (u, *) "  scale =", strfun%scale
               write (u, *) "  p2    =", strfun%s
            case (STRF_ISR)
               call isr_data_write (strfun%isr_data(1), u)
            case (STRF_EPA)
               call epa_data_write (strfun%epa_data(1), u)
            case (STRF_EWA)
               call ewa_data_write (strfun%ewa_data(1), u)
            end select
            call interaction_write &
                 (strfun%int, unit, verbose, show_momentum_sum, show_mass)
         else
            write (u, *) "Structure function setup: [empty]"
         end if
      end subroutine strfun_write
```

Retrieve data

⟨*Strfun: public*⟩+≡
```
   !  public :: strfun_get_name
```

⟨*Strfun: procedures*⟩+≡

```
function strfun_get_name (strfun) result (name)
  type(string_t) :: name
  type(strfun_t), intent(in) :: strfun
  name = strfun%name
end function strfun_get_name
```

Apply structure function

Set kinematics using input random numbers. For some structure functions, we can already compute matrix elements.

⟨*Strfun: public*⟩+≡

```
!  public :: strfun_set_kinematics
```

⟨*Strfun: procedures*⟩+≡

```
subroutine strfun_set_kinematics (strfun, r)
  type(strfun_t), intent(inout) :: strfun
  real(default), dimension(:), intent(in) :: r
  select case (strfun%type)
  case (STRF_LHAPDF)
     call interaction_set_kinematics_lhapdf (strfun%int, &
          strfun%x, strfun%f, strfun%s, r(1), strfun%lhapdf_data(1))
  case (STRF_ISR)
     call interaction_apply_isr (strfun%int, r, strfun%isr_data(1))
  case (STRF_EPA)
     call interaction_apply_epa (strfun%int, r, strfun%epa_data)
  case (STRF_EWA)
     call interaction_apply_ewa (strfun%int, r, strfun%ewa_data)
  end select
end subroutine strfun_set_kinematics
```

Set values where they depend on a separate energy scale:

⟨*Strfun: public*⟩+≡

```
!  public :: strfun_apply
```

⟨*Strfun: procedures*⟩+≡

```
subroutine strfun_apply (strfun, scale)
  type(strfun_t), intent(inout) :: strfun
  real(default), intent(in) :: scale
  strfun%scale = scale
  select case (strfun%type)
  case (STRF_LHAPDF)
     call interaction_apply_lhapdf (strfun%int, scale, &
          strfun%x, strfun%f, strfun%s, strfun%lhapdf_data(1))
  end select
end subroutine strfun_apply
```

10.6.2 Mappings

Definition

Mappings for single structure functions may be defined in the individual sections, but pairwise mappings belong here. We define a mapping type that applies to an array of x parameters identified by their indices. The individual mapping types are identified by a `type` parameter. A mapping may depend on a set on real parameters.

⟨*Strfun: parameters*⟩+≡
```
integer, parameter, public :: SFM_NONE = 0
integer, parameter, public :: SFM_PDFPAIR = 1
integer, parameter, public :: SFM_ISRPAIR = 2
integer, parameter, public :: SFM_EPAPAIR = 3
integer, parameter, public :: SFM_EWAPAIR = 4
```

⟨*Strfun: types*⟩+≡
```
type :: strfun_mapping_t
   private
   integer, dimension(:), allocatable :: index
   integer :: type = SFM_NONE
   real(default), dimension(:), allocatable :: par
end type strfun_mapping_t
```

Initialization:

⟨*Strfun: procedures*⟩+≡
```
subroutine strfun_mapping_init (sf_mapping, index, type, par)
  type(strfun_mapping_t), intent(out) :: sf_mapping
  integer, dimension(:), intent(in) :: index
  integer, intent(in) :: type
  real(default), dimension(:), intent(in) :: par
  allocate (sf_mapping%index (size (index)))
  sf_mapping%index = index
  sf_mapping%type = type
  allocate (sf_mapping%par (size (par)))
  sf_mapping%par = par
end subroutine strfun_mapping_init
```

Output

⟨*Strfun: procedures*⟩+≡
```
subroutine strfun_mapping_write (sf_mapping, unit)
  type(strfun_mapping_t), intent(in) :: sf_mapping
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A)", advance="no") "Strfun mapping for indices: "
  write (u, "(10(1x,I0))")  sf_mapping%index
  write (u, "(1x,A,1x,I0)")  "mapping type =", sf_mapping%type
  write (u, "(1x,A)", advance="no")  "mapping pars ="
  write (u, *)  sf_mapping%par
end subroutine strfun_mapping_write
```

Evaluation

⟨Strfun: procedures⟩+≡
```
subroutine strfun_mapping_apply (sf_mapping, x, factor)
  type(strfun_mapping_t), intent(in) :: sf_mapping
  real(default), dimension(:), intent(inout) :: x
  real(default), intent(inout) :: factor
  real(default), dimension(2) :: x2
  select case (sf_mapping%type)
  case (SFM_PDFPAIR)
     x2 = x(sf_mapping%index)
     call map_unit_square (x2, factor, sf_mapping%par(1))
     x(sf_mapping%index) = x2
  end select
end subroutine strfun_mapping_apply
```

This mapping of the unit square is appropriate in particular for structure functions which are concentrated at the lower end. Instead of a rectangular grid, one set of grid lines corresponds to constant parton c.m. energy. The other set is chosen such that the jacobian is only mildly singular ($\ln x$ which is zero at $x = 1$), corresponding to an initial concentration of sampling points at the maximum energy. If `power` is greater than one (the default), points are also concentrated at the lower end.

⟨Strfun: procedures⟩+≡
```
subroutine map_unit_square (x, factor, power)
  real(kind=default), dimension(2), intent(inout) :: x
  real(kind=default), intent(inout) :: factor
  real(kind=default), intent(in), optional :: power
  real(kind=default) :: xx, yy
  xx = x(1)
  yy = x(2)
  if (present(power)) then
     if (x(1) > 0 .and. power > 1) then
        xx = x(1)**power
        factor = factor * power * xx / x(1)
     end if
  end if
  if (xx /= 0) then
     x(1) = xx ** yy
     x(2) = xx / x(1)
     factor = factor * abs (log (xx))
  else
     x = 0
  end if
end subroutine map_unit_square
```

10.6.3 Structure function chains

Definition

The structure function chain contains an array of structure functions, where each one has one or more free parameters. For each structure function there is an interaction, an image of the interaction within the `strfun` object, which is needed when quantum numbers are reduced. Furthermore, an array of evaluators which cumulatively multiply the structure functions. The last evaluator is connected to the hard matrix element.

The `last_strfun` and `out_index` index (pairs) identify, for each beam, the last structure function and the outgoing particle. The `coll_index` (pair) identifies the outgoing particles in the last evaluator.

For a decay, this structure is also used, but normally there are no structure functions besides the "beam" object.

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_t
```

⟨*Strfun: types*⟩+≡
```
    type :: strfun_chain_t
      private
      type(beam_t) :: beam
      integer :: n_strfun = 0
      integer :: n_mapping = 0
      type(strfun_t), dimension(:), allocatable :: strfun
      type(strfun_mapping_t), dimension(:), allocatable :: sf_mapping
      real(default) :: mapping_factor = 1
      integer :: n_parameters_tot = 0
      integer, dimension(:), allocatable :: n_parameters
      type(evaluator_t), dimension(:), allocatable :: eval
      integer, dimension(:), allocatable :: last_strfun
      integer, dimension(:), allocatable :: out_index
      integer, dimension(:), allocatable :: coll_index
    end type strfun_chain_t
```

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_init
```

⟨*Strfun: procedures*⟩+≡
```
    subroutine strfun_chain_init (sfchain, beam_data, n_strfun, n_mapping)
      type(strfun_chain_t), intent(out) :: sfchain
      type(beam_data_t), intent(in), target :: beam_data
      integer, intent(in) :: n_strfun, n_mapping
      integer :: i
      sfchain%n_strfun = n_strfun
      allocate (sfchain%strfun (n_strfun))
      allocate (sfchain%sf_mapping (n_mapping))
      allocate (sfchain%n_parameters (n_strfun))
      sfchain%n_parameters = 0
      allocate (sfchain%eval (n_strfun))
      call beam_init (sfchain%beam, beam_data)
      allocate (sfchain%last_strfun (beam_data%n))
      allocate (sfchain%out_index (beam_data%n))
      allocate (sfchain%coll_index (beam_data%n))
```

```
      sfchain%last_strfun = 0
      do i = 1, size (sfchain%out_index)
         sfchain%out_index(i) = i
         sfchain%coll_index(i) = i
      end do
    end subroutine strfun_chain_init
```

Set beam momenta directly without changing anything else.
⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_set_beam_momenta
```

⟨*Strfun: procedures*⟩+≡
```
    subroutine strfun_chain_set_beam_momenta (sfchain, p)
      type(strfun_chain_t), intent(inout) :: sfchain
      type(vector4_t), dimension(:), intent(in) :: p
      call beam_set_momenta (sfchain%beam, p)
    end subroutine strfun_chain_set_beam_momenta
```

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_final
```

⟨*Strfun: procedures*⟩+≡
```
    subroutine strfun_chain_final (sfchain)
      type(strfun_chain_t), intent(inout) :: sfchain
      call beam_final (sfchain%beam)
      if (allocated (sfchain%strfun))  call strfun_final (sfchain%strfun)
      if (allocated (sfchain%eval))  call evaluator_final (sfchain%eval)
    end subroutine strfun_chain_final
```

I/O

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_write
```

⟨*Strfun: procedures*⟩+≡
```
    subroutine strfun_chain_write &
        (sfchain, unit, verbose, show_momentum_sum, show_mass)
      type(strfun_chain_t), intent(in) :: sfchain
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
      integer :: u, i
      logical :: verb
      verb = .false.;  if (present (verbose))  verb = verbose
      u = output_unit (unit);  if (u < 0)  return
      write (u, *)  "Structure function chain:"
      write (u, *)
      call beam_write (sfchain%beam, unit, verbose, show_momentum_sum, show_mass)
      if (allocated (sfchain%strfun)) then
         do i = 1, size (sfchain%strfun)
            write (u, *)
            call strfun_write &
                 (sfchain%strfun(i), unit, verbose, show_momentum_sum, show_mass)
            write (u, *)  "number of parameters = ", sfchain%n_parameters(i)
```

```
         end do
      end if
      if (allocated (sfchain%sf_mapping)) then
         do i = 1, size (sfchain%sf_mapping)
            write (u, *)
            call strfun_mapping_write (sfchain%sf_mapping(i), unit)
         end do
      end if
      if (allocated (sfchain%eval)) then
         write (u, *)
         write (u, *) "Evaluators:"
         do i = 1, size (sfchain%eval)
            call evaluator_write &
                 (sfchain%eval(i), unit, verbose, show_momentum_sum, show_mass)
         end do
      end if
      write (u, *)
      write (u, *)  "Total number of parameters      = ", &
           sfchain%n_parameters_tot
      write (u, "(1x,A)", advance="no")  "Last structure function (index) = "
      if (allocated (sfchain%last_strfun)) then
         write (u, *) sfchain%last_strfun
      else
         write (u, *) "[not allocated]"
      end if
      write (u, "(1x,A)", advance="no")  "Outgoing particles (index)      = "
      if (allocated (sfchain%out_index)) then
         write (u, *) sfchain%out_index
      else
         write (u, *) "[not allocated]"
      end if
      write (u, "(1x,A)", advance="no")  "Colliding particles (index)     = "
      if (allocated (sfchain%coll_index)) then
         write (u, *)  sfchain%coll_index
      else
         write (u, *) "[not allocated]"
      end if
    end subroutine strfun_chain_write
```

Defined assignment

Deep copy of all components.

⟨*Strfun: public*⟩+≡
```
    public :: assignment(=)
```

⟨*Strfun: interfaces*⟩+≡
```
    interface assignment(=)
       module procedure strfun_chain_assign
    end interface
```

⟨*Strfun: procedures*⟩+≡
```
    subroutine strfun_chain_assign (sfchain_out, sfchain_in)
      type(strfun_chain_t), intent(out) :: sfchain_out
```

```
type(strfun_chain_t), intent(in) :: sfchain_in
sfchain_out%beam = sfchain_in%beam
sfchain_out%n_strfun = sfchain_in%n_strfun
sfchain_out%n_mapping = sfchain_in%n_mapping
if (allocated (sfchain_in%strfun)) then
   allocate (sfchain_out%strfun (size (sfchain_in%strfun)))
   sfchain_out%strfun = sfchain_in%strfun
end if
if (allocated (sfchain_in%sf_mapping)) then
   allocate (sfchain_out%sf_mapping (size (sfchain_in%sf_mapping)))
   sfchain_out%sf_mapping = sfchain_in%sf_mapping
end if
sfchain_out%mapping_factor = sfchain_in%mapping_factor
sfchain_out%n_parameters_tot = sfchain_in%n_parameters_tot
if (allocated (sfchain_in%n_parameters)) then
   allocate (sfchain_out%n_parameters (size (sfchain_in%n_parameters)))
   sfchain_out%n_parameters = sfchain_in%n_parameters
end if
if (allocated (sfchain_in%eval)) then
   allocate (sfchain_out%eval (size (sfchain_in%eval)))
   sfchain_out%eval = sfchain_in%eval
end if
if (allocated (sfchain_in%last_strfun)) then
   allocate (sfchain_out%last_strfun (size (sfchain_in%last_strfun)))
   sfchain_out%last_strfun = sfchain_in%last_strfun
end if
if (allocated (sfchain_in%out_index)) then
   allocate (sfchain_out%out_index (size (sfchain_in%out_index)))
   sfchain_out%out_index = sfchain_in%out_index
end if
if (allocated (sfchain_in%coll_index)) then
   allocate (sfchain_out%coll_index (size (sfchain_in%coll_index)))
   sfchain_out%coll_index = sfchain_in%coll_index
end if
end subroutine strfun_chain_assign
```

Accessing contents

The number of active structure functions.

⟨*Strfun: public*⟩+≡

```
public :: strfun_chain_get_n_strfun
```

⟨*Strfun: procedures*⟩+≡

```
function strfun_chain_get_n_strfun (sfchain) result (n)
  integer :: n
  type(strfun_chain_t), intent(in) :: sfchain
  n = sfchain%n_strfun
end function strfun_chain_get_n_strfun
```

The number of free parameters (x values) needed for evaluating the structure functions.

⟨*Strfun: public*⟩+≡

```
public :: strfun_chain_get_n_parameters_tot
```

⟨*Strfun: procedures*⟩+≡
```
    function strfun_chain_get_n_parameters_tot (sfchain) result (n)
      integer :: n
      type(strfun_chain_t), intent(in) :: sfchain
      n = sfchain%n_parameters_tot
    end function strfun_chain_get_n_parameters_tot
```

The number of virtual particles in the structure function evaluators. These are the beam particles plus all particles that do not appear as outgoing.

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_get_n_vir
```

⟨*Strfun: procedures*⟩+≡
```
    function strfun_chain_get_n_vir (sfchain) result (n)
      integer :: n
      type(strfun_chain_t), intent(in) :: sfchain
      if (sfchain%n_strfun /= 0) then
         n = evaluator_get_n_vir (sfchain%eval(sfchain%n_strfun))
      else
         n = 0
      end if
    end function strfun_chain_get_n_vir
```

Return any extra factor resulting from explicit mappings of the x parameters.

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_get_mapping_factor
```

⟨*Strfun: procedures*⟩+≡
```
    function strfun_chain_get_mapping_factor (sfchain) result (f)
      real(default) :: f
      type(strfun_chain_t), intent(in) :: sfchain
      f = sfchain%mapping_factor
    end function strfun_chain_get_mapping_factor
```

For pseudo-structure functions that actually are an external generator, the integration region must not be mapped and stay rigid. This function returns an array which tells, for each integration parameter, whether the corresponding integration dimension is rigid.

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_dimension_is_rigid
```

⟨*Strfun: procedures*⟩+≡
```
    function strfun_chain_dimension_is_rigid (sfchain) result (rigid)
      logical, dimension(:), allocatable :: rigid
      type(strfun_chain_t), intent(in) :: sfchain
      integer :: i, j, k
      allocate (rigid (sfchain%n_parameters_tot))
      k = 0
      do i = 1, size (sfchain%n_parameters)
         do j = 1, sfchain%n_parameters(i)
            k = k + 1
            select case (sfchain%strfun(i)%type)
            case default
```

```
              rigid(k) = .false.
          end select
        end do
      end do
    end function strfun_chain_dimension_is_rigid
```

Return the indices of the colliding particles.

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_get_colliding_particles
```

⟨*Strfun: procedures*⟩+≡
```
    function strfun_chain_get_colliding_particles (sfchain) result (index)
      integer, dimension(:), allocatable :: index
      type(strfun_chain_t), intent(in) :: sfchain
      allocate (index (size (sfchain%coll_index)))
      index = sfchain%coll_index
    end function strfun_chain_get_colliding_particles
```

Return the quantum-numbers mask for the colliding particles. This is extracted
from the last evaluator in the chain.

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_get_colliding_particles_mask
```

⟨*Strfun: procedures*⟩+≡
```
    function strfun_chain_get_colliding_particles_mask (sfchain) result (mask)
      type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
      type(strfun_chain_t), intent(in), target :: sfchain
      integer :: n_strfun
      type(quantum_numbers_mask_t), dimension(:), allocatable :: mask_eval
      allocate (mask (size (sfchain%coll_index)))
      n_strfun = sfchain%n_strfun
      if (n_strfun /= 0) then
         allocate (mask_eval (evaluator_get_n_tot (sfchain%eval(n_strfun))))
         mask_eval = evaluator_get_mask (sfchain%eval(n_strfun))
         mask = mask_eval(sfchain%coll_index)
      else
         mask = interaction_get_mask (beam_get_int_ptr (sfchain%beam))
      end if
    end function strfun_chain_get_colliding_particles_mask
```

Return a pointer to the beam interaction.

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_get_beam_int_ptr
```

⟨*Strfun: procedures*⟩+≡
```
    function strfun_chain_get_beam_int_ptr (sfchain) result (int)
      type(interaction_t), pointer :: int
      type(strfun_chain_t), intent(in), target :: sfchain
      int => beam_get_int_ptr (sfchain%beam)
    end function strfun_chain_get_beam_int_ptr
```

Return a pointer to the last evaluator, which wraps up all structure functions.

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_get_last_evaluator_ptr
```

⟨*Strfun: procedures*⟩+≡
```
    function strfun_chain_get_last_evaluator_ptr (sfchain) result (eval)
      type(evaluator_t), pointer :: eval
      type(strfun_chain_t), intent(in), target :: sfchain
      if (sfchain%n_strfun /= 0) then
         eval => sfchain%eval(sfchain%n_strfun)
      else
         eval => null ()
      end if
    end function strfun_chain_get_last_evaluator_ptr
```

Setting up structure functions

The index `i` is the overall structure function counter. `line` indicates the beam(s) for which the structure function applies, either 1 or 2, or 0 for both beams.

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_set_strfun
```

⟨*Strfun: interfaces*⟩+≡
```
    interface strfun_chain_set_strfun
       module procedure strfun_chain_set_lhapdf
       module procedure strfun_chain_set_isr
       module procedure strfun_chain_set_epa
       module procedure strfun_chain_set_ewa
    end interface
```

⟨*Strfun: procedures*⟩+≡
```
    subroutine strfun_chain_set_lhapdf &
         (sfchain, i, line, lhapdf_data, n_parameters)
      type(strfun_chain_t), intent(inout), target :: sfchain
      integer, intent(in) :: i, line, n_parameters
      type(lhapdf_data_t), intent(in) :: lhapdf_data
      call strfun_init (sfchain%strfun(i), lhapdf_data)
      sfchain%n_parameters(i) = n_parameters
      call strfun_chain_link (sfchain, i, line, (/1/), (/3/))
    end subroutine strfun_chain_set_lhapdf

    subroutine strfun_chain_set_isr &
         (sfchain, i, line, isr_data, n_parameters)
      type(strfun_chain_t), intent(inout), target :: sfchain
      integer, intent(in) :: i, line, n_parameters
      type(isr_data_t), intent(in) :: isr_data
      call strfun_init (sfchain%strfun(i), isr_data)
      sfchain%n_parameters(i) = n_parameters
      call strfun_chain_link (sfchain, i, line, (/1/), (/3/))
    end subroutine strfun_chain_set_isr

    subroutine strfun_chain_set_epa &
         (sfchain, i, line, epa_data, n_parameters)
      type(strfun_chain_t), intent(inout), target :: sfchain
      integer, intent(in) :: i, line, n_parameters
      type(epa_data_t), intent(in) :: epa_data
      call strfun_init (sfchain%strfun(i), epa_data)
```

```
      sfchain%n_parameters(i) = n_parameters
      call strfun_chain_link (sfchain, i, line, (/1/), (/3/))
    end subroutine strfun_chain_set_epa

    subroutine strfun_chain_set_ewa &
         (sfchain, i, line, ewa_data, n_parameters, id)
      type(strfun_chain_t), intent(inout), target :: sfchain
      integer, intent(in) :: i, line, n_parameters, id
      type(ewa_data_t), intent(inout) :: ewa_data
      call strfun_init (sfchain%strfun(i), ewa_data, id)
      sfchain%n_parameters(i) = n_parameters
      call strfun_chain_link (sfchain, i, line, (/1/), (/3/))
    end subroutine strfun_chain_set_ewa
```

This procedure links a new structure function to the existing chain. `i` is the overall structure function counter. `line` indicates the beam(s) to which the structure function applies; 0 is for both beams. The last two arguments are the indices of the incoming and outgoing particle(s) within the current structure function. For a single-beam (double-beam) structure function, these arrays are of length 1 (2), respectively.

The connections of outgoing/incoming particles are recorded as links in the new structure-function entry `sfchain%strfun(i)`.

⟨*Strfun: procedures*⟩+≡
```
    subroutine strfun_chain_link (sfchain, i, line, in_index, out_index)
      type(strfun_chain_t), intent(inout), target :: sfchain
      integer, intent(in) :: i, line
      integer, dimension(:), intent(in) :: in_index, out_index
      select case (line)
      case (0)
         call link_single (1, in_index(1))
         call link_single (2, in_index(2))
         sfchain%last_strfun = i
         sfchain%out_index = out_index
      case default
         call link_single (line, in_index(1))
         sfchain%last_strfun(line) = i
         sfchain%out_index(line) = out_index(1)
      end select
    contains
      subroutine link_single (line, in_index)
        integer, intent(in) :: line, in_index
        integer :: j
        j = sfchain%last_strfun(line)
        select case (j)
        case (0)
           call interaction_set_source_link &
                (sfchain%strfun(i)%int, in_index, &
                 sfchain%beam, sfchain%out_index(line))
        case default
           call interaction_set_source_link &
                (sfchain%strfun(i)%int, in_index, &
                 sfchain%strfun(j)%int, sfchain%out_index(j))
        end select
```

```
      end subroutine link_single
    end subroutine strfun_chain_link
```

Setting up mappings

Set a particular mapping with a known type.

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_set_mapping
```

⟨*Strfun: procedures*⟩+≡
```
    subroutine strfun_chain_set_mapping (sfchain, i, index, type, par)
      type(strfun_chain_t), intent(inout) :: sfchain
      integer, intent(in) :: i
      integer, dimension(:), intent(in) :: index
      integer, intent(in) :: type
      real(default), dimension(:), intent(in) :: par
      call strfun_mapping_init (sfchain%sf_mapping(i), index, type, par)
    end subroutine strfun_chain_set_mapping
```

Evaluators

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_make_evaluators
```

⟨*Strfun: procedures*⟩+≡
```
    subroutine strfun_chain_make_evaluators (sfchain, ok)
      type(strfun_chain_t), intent(inout), target :: sfchain
      logical, intent(out), optional :: ok
      type(interaction_t), pointer :: beam_int, eval_int
      type(quantum_numbers_mask_t) :: qn_mask_conn
      type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask_beam
      integer :: i, j
      sfchain%n_parameters_tot = sum (sfchain%n_parameters)
      beam_int => beam_get_int_ptr (sfchain%beam)
      if (.not. associated (beam_int))  call msg_bug &
           ("strfun_chain_make_evaluators: null beam pointer")
      allocate (qn_mask_beam (interaction_get_n_out (beam_int)))
      qn_mask_beam = interaction_get_mask (beam_int)
      call interaction_exchange_mask (beam_int)
      do i = 1, size (sfchain%strfun) - 1
         call interaction_exchange_mask (sfchain%strfun(i)%int)
      end do
      do i = size (sfchain%strfun), 1, -1
         call interaction_exchange_mask (sfchain%strfun(i)%int)
      end do
      if (any (qn_mask_beam .neqv. interaction_get_mask (beam_int))) then
         call beam_write (sfchain%beam)
         call msg_fatal (" Beam polarization/color/flavor incompatible with structure functions")
      end if
      eval_int => beam_int
      do i = 1, size (sfchain%strfun)
         qn_mask_conn = new_quantum_numbers_mask (.false., .false., .true.)
```

```
            call evaluator_init_product (sfchain%eval(i), eval_int, &
                sfchain%strfun(i)%int, qn_mask_conn)
            if (evaluator_is_empty (sfchain%eval(i))) then
                call msg_fatal ("Mismatch in beam and structure-function chain")
                if (present (ok))  ok = .false.
                return
            end if
            eval_int => evaluator_get_int_ptr (sfchain%eval(i))
            do j = 1, size (sfchain%coll_index)
                sfchain%coll_index(i) = interaction_find_link (eval_int, &
                    sfchain%strfun(sfchain%last_strfun(i))%int, &
                    sfchain%out_index(i))
            end do
            if (any (sfchain%coll_index == 0)) &
                call msg_bug ("Structure functions: " &
                // "colliding particles can't be determined")
        end do
        if (present (ok))  ok = .true.
    end subroutine strfun_chain_make_evaluators
```

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_set_kinematics
```

⟨*Strfun: procedures*⟩+≡
```
    subroutine strfun_chain_set_kinematics (sfchain, r)
      type(strfun_chain_t), intent(inout) :: sfchain
      real(default), dimension(:), intent(in) :: r
      real(default), dimension(size(r)) :: x
      integer :: i, n, n1
      if (size (r) == sfchain%n_parameters_tot) then
          x = r
          sfchain%mapping_factor = 1
          do i = 1, size (sfchain%sf_mapping)
              call strfun_mapping_apply &
                    (sfchain%sf_mapping(i), x, sfchain%mapping_factor)
          end do
          n = 0
          do i = 1, size (sfchain%strfun)
              call interaction_receive_momenta (sfchain%strfun(i)%int)
              n1 = sfchain%n_parameters(i)
              call strfun_set_kinematics (sfchain%strfun(i), x(n+1:n+n1))
              n = n + n1
          end do
          do i = 1, size (sfchain%strfun)
              call evaluator_receive_momenta (sfchain%eval(i))
          end do
      else
          call msg_bug ("Structure functions: mismatch in number of parameters")
      end if
    end subroutine strfun_chain_set_kinematics
```

⟨*Strfun: public*⟩+≡
```
    public :: strfun_chain_evaluate
```

⟨*Strfun: procedures*⟩+≡

```
subroutine strfun_chain_evaluate (sfchain, scale)
  type(strfun_chain_t), intent(inout) :: sfchain
  real(default), intent(in) :: scale
  integer :: i
  do i = size (sfchain%strfun), 1, -1
     call strfun_apply (sfchain%strfun(i), scale)
  end do
  do i = 1, size (sfchain%eval)
     call evaluator_evaluate (sfchain%eval(i))
  end do
end subroutine strfun_chain_evaluate
```

10.6.4 Test

⟨*Strfun: public*⟩+≡
```
    public :: strfun_test
```

⟨*Strfun: procedures*⟩+≡
```
    subroutine strfun_test ()
      use os_interface, only: os_data_t
      type(os_data_t) :: os_data
      type(model_t), pointer :: model
      print *, "*** Read model file"
      call syntax_model_file_init ()
      call model_list_read_model &
            (var_str("QCD"), var_str("test.mdl"), os_data, model)
      call syntax_model_file_final ()
      print *, "********************************************************"
      call isr_test (model)
      print *, "********************************************************"
      call epa_test (model)
      print *, "********************************************************"
      call lhapdf_test (model)
    end subroutine strfun_test

    subroutine isr_test (model)
      use flavors
      use polarizations
      type(model_t), intent(in), target :: model
      type(flavor_t), dimension(2) :: flv
      type(polarization_t), dimension(2) :: pol
      type(beam_data_t), target :: beam_data
      type(isr_data_t), dimension(2) :: isr_data
      type(strfun_chain_t), target :: sfchain
      integer :: i
      print *, "*** ISR test"
      call flavor_init (flv, (/11, -11/), model)
      call polarization_init_unpolarized (pol(1), flv(1))
      call polarization_init_unpolarized (pol(2), flv(2))
      call beam_data_init_sqrts (beam_data, 500._default, flv, pol)
      do i = 1, 2
         call isr_data_init (isr_data(i), &
               model, flv(i), 0.06_default, 500._default, 0.511e-3_default)
```

```
      end do
      call strfun_chain_init (sfchain, beam_data, 2, 0)
      call strfun_chain_set_strfun (sfchain, 1, 1, isr_data(1), 1)
      call strfun_chain_set_strfun (sfchain, 2, 2, isr_data(2), 3)
      call strfun_chain_make_evaluators (sfchain)
      call strfun_chain_set_kinematics &
           (sfchain, (/0.8_default, 0.4_default, 0.5_default, 0.2_default/))
      call strfun_chain_evaluate (sfchain, 0._default)
      call strfun_chain_write (sfchain)
      call strfun_chain_final (sfchain)
    end subroutine isr_test

    subroutine epa_test (model)
      use flavors
      use polarizations
      type(model_t), intent(in), target :: model
      type(flavor_t), dimension(2) :: flv
      type(polarization_t), dimension(2) :: pol
      type(beam_data_t) :: beam_data
      type(epa_data_t) :: epa_data1
      type(epa_data_t) :: epa_data2
      type(strfun_chain_t), target :: sfchain
      print *, "*** EPA test"
      call flavor_init (flv, (/2, 1/), model)
      ! Prepare beams
      call polarization_init_circular (pol(1), flv(1), 0.3_default)
      call polarization_init_unpolarized (pol(2), flv(2))
      call beam_data_init_sqrts (beam_data, 1000._default, flv, pol)
      call strfun_chain_init (sfchain, beam_data, 2, 0)
      ! Initialize EPA for both
      call epa_data_init (epa_data1, model, &
           flv(1), 0.06_default, 1.e-6_default, 0._default, 500._default, &
           511.e-6_default)
      call epa_data_init (epa_data2, model, &
           flv(2), 0.06_default, 1.e-6_default, 1._default, 500._default)
      call strfun_chain_set_strfun (sfchain, 1, 1, epa_data1, 1)
      call strfun_chain_set_strfun (sfchain, 2, 2, epa_data2, 3)
!     call strfun_chain_write (sfchain); stop
      call strfun_chain_make_evaluators (sfchain)
      call strfun_chain_set_kinematics &
           (sfchain, (/0.8_default, 0.4_default, 0.5_default, 0.2_default/))
      call strfun_chain_evaluate (sfchain, 0._default)
      call strfun_chain_write (sfchain)
      ! Clean up
      call beam_data_final (beam_data)
      call polarization_final (pol)
      call strfun_chain_final (sfchain)
    end subroutine epa_test

    subroutine lhapdf_test (model)
      use flavors
      use polarizations
      type(model_t), intent(in), target :: model
      type(beam_data_t) :: beam_data
```

```
      type(flavor_t), dimension(2) :: flv
      type(polarization_t), dimension(2) :: pol
      type(lhapdf_data_t), dimension(2) :: data
      type(lhapdf_status_t) :: lhapdf_status
      type(strfun_chain_t), target :: sfchain
      real(default) :: scale
      print *, "*** LHAPDF test"
      call flavor_init (flv, (/ -PROTON, PHOTON /), model)
      call polarization_init_unpolarized (pol(1), flv(1))
      call polarization_init_unpolarized (pol(2), flv(2))
      call beam_data_init_sqrts (beam_data, 2000._default, flv, pol)
      call strfun_chain_init (sfchain, beam_data, 2, 0)
      call lhapdf_data_init (data(1), lhapdf_status, model, flv(1), member=1)
      call lhapdf_data_init (data(2), lhapdf_status, model, flv(2), &
           file=var_str("SASG.LHgrid"), photon_scheme=1)
      call lhapdf_data_set_mask (data(2), &
           (/.false.,.false.,.false., .true., .true., .true., &
             .false., &
             .true., .true., .true., .false., .false., .false. /))
!     call strfun_chain_write (sfchain); stop
      call strfun_chain_set_strfun (sfchain, 1, 1, data(1), 1)
      call strfun_chain_set_strfun (sfchain, 2, 2, data(2), 1)
!     call strfun_chain_write (sfchain); stop
      call strfun_chain_make_evaluators (sfchain)
      call strfun_chain_set_kinematics (sfchain, (/0.9_default, 0.4_default/))
      scale = 1.e3_default
      call strfun_chain_evaluate (sfchain, scale)
      call strfun_chain_write (sfchain)
      call strfun_chain_final (sfchain)
  end subroutine lhapdf_test
```

Chapter 11

Partonic Events

This chapter deals with and combines the various components (interactions) of a partonic event: beams, spectra, scattering, decays.

Chapter 12

Phase space and hard matrix elements

These modules contain the internal representation and evaluation of phase space and the interface to (hard-)process evaluation.

mappings Generate invariant masses and decay angles from given random numbers (or the inverse operation). Each mapping pertains to a particular node in a phase-space tree. Different mappings account for uniform distributions, resonances, zero-mass behavior, and so on.

phs_trees Phase space parameterizations for scattering processes are defined recursively as if there was an initial particle decaying. This module sets up a representation in terms of abstract trees, where each node gets a unique binary number. Each tree is stored as an array of branches, where integers indicate the connections. This emulates pointers in a transparent way. Real pointers would also be possible, but seem to be less efficient for this particular case.

phs_forests The type defined by this module collects the decay trees corresponding to a given process and the applicable mappings. To set this up, a file is read which is either written by the user or by the **cascades** module functions. The module also contains the routines that evaluate phase space, i.e., generate momenta from random numbers and back.

cascades This module is a Feynman diagram generator with the particular purpose of finding the phase space parameterizations best suited for a given process. It uses a model file to set up the possible vertices, generates all possible diagrams, identifies resonances and singularities, and simplifies the list by merging equivalent diagrams and dropping irrelevant ones. This process can be controlled at several points by user-defined parameters. Note that it depends on the particular values of particle masses, so it cannot be done before reading the input file.

12.1 Mappings

Mappings are objects that encode the transformation of the interval $(0, 1)$ to a physical variable m^2 or $\cos\theta$ (and back), as it is used in the phase space parameterization. The mapping objects contain fixed parameters, the associated methods implement the mapping and inverse mapping operations, including the computation of the Jacobian (phase space factor).

⟨`mappings.f90`⟩≡
  ⟨*File header*⟩

  ```
  module mappings
  ```

  ⟨*Use kinds*⟩
  ```
    use kinds, only: TC !NODEP!
  ```
  ⟨*Use strings*⟩
  ```
    use constants, only: pi !NODEP!
  ```
  ⟨*Use file utils*⟩
  ```
    use diagnostics !NODEP!
    use md5
    use models
    use flavors
  ```

  ⟨*Standard module head*⟩

  ⟨*Mappings: public*⟩

  ⟨*Mappings: parameters*⟩

  ⟨*Mappings: types*⟩

  ⟨*Mappings: interfaces*⟩

  ```
  contains
  ```

  ⟨*Mappings: procedures*⟩

  ```
  end module mappings
  ```

12.1.1 Default parameters

This type holds the default parameters, needed for setting the scale in cases where no mass parameter is available. The contents are public.

⟨*Mappings: public*⟩≡
  ```
    public :: mapping_defaults_t
  ```

⟨*Mappings: types*⟩≡
  ```
    type :: mapping_defaults_t
       real(default) :: energy_scale = 10
       real(default) :: invariant_mass_scale = 10
       real(default) :: momentum_transfer_scale = 10
    end type mapping_defaults_t
  ```

⟨*Mappings: public*⟩+≡
```
    public :: mapping_defaults_md5sum
```

⟨*Mappings: procedures*⟩≡
```
    function mapping_defaults_md5sum (mapping_defaults) result (md5sum_map)
      character(32) :: md5sum_map
      type(mapping_defaults_t), intent(in) :: mapping_defaults
      integer :: u
      u = free_unit ()
      open (u, status = "scratch")
      write (u, *)  mapping_defaults%energy_scale
      write (u, *)  mapping_defaults%invariant_mass_scale
      write (u, *)  mapping_defaults%momentum_transfer_scale
      rewind (u)
      md5sum_map = md5sum (u)
      close (u)
    end function mapping_defaults_md5sum
```

12.1.2 The Mapping type

Each mapping has a type (e.g., s-channel, infrared), a binary code (redundant, but useful for debugging), and a reference particle. The flavor code of this particle is stored for bookkeeping reasons, what matters are the mass and width of this particle. Furthermore, depending on the type, various mapping parameters can be set and used.

The parameters `a1` to `a3` (for m^2 mappings) and `b1` to `b3` (for $\cos\theta$ mappings) are values that are stored once to speed up the calculation, if `variable_limits` is false. The exact meaning of these parameters depends on the mapping type. The limits are fixed if there is a fixed c.m. energy.

⟨*Mappings: public*⟩+≡
```
    public :: mapping_t
```

⟨*Mappings: types*⟩+≡
```
    type :: mapping_t
      private
      integer :: type = NO_MAPPING
      integer(TC) :: bincode
      type(flavor_t) :: flv
      real(default) :: mass = 0
      real(default) :: width = 0
      logical :: a_unknown = .true.
      real(default) :: a1, a2, a3
      logical :: b_unknown = .true.
      real(default) :: b1, b2, b3
      logical :: variable_limits = .true.
    end type mapping_t
```

The valid mapping types:

⟨*Mappings: parameters*⟩≡
  ⟨*Mapping modes*⟩

12.1.3 Screen output

Do not write empty mappings.

⟨*Mappings: public*⟩+≡
```
public :: mapping_write
```

⟨*Mappings: procedures*⟩+≡
```
subroutine mapping_write (map, unit)
  type(mapping_t), intent(in) :: map
  integer, intent(in), optional :: unit
  integer :: u
  character(len=9) :: str
  u = output_unit (unit);  if (u < 0)  return
  select case(map%type)
  case(S_CHANNEL); str = "s_channel"
  case(COLLINEAR); str = "collinear"
  case(INFRARED);  str = "infrared "
  case(RADIATION); str = "radiation"
  case(T_CHANNEL); str = "t_channel"
  case(U_CHANNEL); str = "u_channel"
  end select
  if (map%type /= NO_MAPPING) then
     write (u, '(1x,A,I4,A)') &
          "Branch #", map%bincode, ":  " // &
          "Mapping (" // str // ") for particle " // &
          '"' // char (flavor_get_name (map%flv)) // '"'
  end if
end subroutine mapping_write
```

12.1.4 Define a mapping

The initialization routine sets the mapping type and the particle (binary code
and flavor code) for which the mapping applies (e.g., a Z resonance in branch
#3). We only need the absolute value of the flavor code.

⟨*Mappings: public*⟩+≡
```
public :: mapping_init
```

⟨*Mappings: procedures*⟩+≡
```
subroutine mapping_init (mapping, bincode, type, f, model)
  type(mapping_t), intent(inout) :: mapping
  integer(TC), intent(in) :: bincode
  type(string_t), intent(in) :: type
  integer, intent(in) :: f
  type(model_t), intent(in), target :: model
  mapping%bincode = bincode
  select case (char (type))
  case ("s_channel");  mapping%type = S_CHANNEL
  case ("collinear");  mapping%type = COLLINEAR
  case ("infrared");   mapping%type = INFRARED
  case ("radiation");  mapping%type = RADIATION
  case ("t_channel");  mapping%type = T_CHANNEL
  case ("u_channel");  mapping%type = U_CHANNEL
  end select
```

```
      call flavor_init (mapping%flv, abs (f), model)
   end subroutine mapping_init
```

This sets the actual mass and width, using a parameter set. Since the auxiliary
parameters will only be determined when the mapping is first called, they are
marked as unknown.

⟨*Mappings: public*⟩+≡
```
   public :: mapping_set_parameters
```

⟨*Mappings: procedures*⟩+≡
```
   subroutine mapping_set_parameters (map, mapping_defaults, variable_limits)
     type(mapping_t), intent(inout) :: map
     type(mapping_defaults_t), intent(in) :: mapping_defaults
     logical, intent(in) :: variable_limits
     if (map%type /= NO_MAPPING) then
        map%mass  = flavor_get_mass (map%flv)
        map%width = flavor_get_width (map%flv)
        map%variable_limits = variable_limits
        map%a_unknown = .true.
        map%b_unknown = .true.
        select case (map%type)
        case (S_CHANNEL)
           if (map%mass <= 0) then
              call mapping_write (map)
              call msg_fatal &
                   & (" S-channel resonance must have positive mass")
           else if (map%width <= 0) then
              call mapping_write (map)
              call msg_fatal &
                   & (" S-channel resonance must have positive width")
           end if
        case (RADIATION)
           map%width = max (map%width, mapping_defaults%energy_scale)
        case (INFRARED, COLLINEAR)
           map%mass = max (map%mass, mapping_defaults%invariant_mass_scale)
        case (T_CHANNEL, U_CHANNEL)
           map%mass = max (map%mass, mapping_defaults%momentum_transfer_scale)
        end select
     end if
   end subroutine mapping_set_parameters
```

12.1.5 Compare mappings

Equality for single mappings and arrays

⟨*Mappings: public*⟩+≡
```
   public :: operator(==)
```

⟨*Mappings: interfaces*⟩≡
```
   interface operator(==)
      module procedure mapping_equal
   end interface
```

⟨*Mappings: procedures*⟩+≡
```
function mapping_equal (m1, m2) result (equal)
  type(mapping_t), intent(in) :: m1, m2
  logical :: equal
  if (m1%type == m2%type) then
     select case (m1%type)
     case (NO_MAPPING)
        equal = .true.
     case (S_CHANNEL, RADIATION)
        equal = (m1%mass == m2%mass) .and. (m1%width == m2%width)
     case default
        equal = (m1%mass == m2%mass)
     end select
  else
     equal = .false.
  end if
end function mapping_equal
```

12.1.6 Mappings of the invariant mass

Inserting an x value between 0 and 1, we want to compute the corresponding invariant mass $m^2(x)$ and the jacobian, aka phase space factor $f(x)$. We also need the reverse operation.

In general, the phase space factor f is defined by

$$\frac{1}{s} \int_{m^2_{\min}}^{m^2_{\max}} dm^2\, g(m^2) = \int_0^1 dx\, \frac{1}{s} \frac{dm^2}{dx}\, g(m^2(x)) = \int_0^1 dx\, f(x)\, g(x), \quad (12.1)$$

where thus

$$f(x) = \frac{1}{s} \frac{dm^2}{dx}. \quad (12.2)$$

With this mapping, a function of the form

$$g(m^2) = c\frac{dx(m^2)}{dm^2} \quad (12.3)$$

is mapped to a constant:

$$\frac{1}{s} \int_{m^2_{\min}}^{m^2_{\max}} dm^2\, g(m^2) = \int_0^1 dx\, f(x)\, g(m^2(x)) = \int_0^1 dx\, \frac{c}{s}. \quad (12.4)$$

Here is the mapping routine. Input are the available energy squared `s`, the limits for m^2, and the x value. Output are the m^2 value and the phase space factor f.

⟨*Mappings: public*⟩+≡
```
public :: mapping_compute_msq_from_x
```

⟨*Mappings: procedures*⟩+≡
```
subroutine mapping_compute_msq_from_x (map, s, msq_min, msq_max, msq, f, x)
  type(mapping_t), intent(inout) :: map
  real(default), intent(in) :: s, msq_min, msq_max
  real(default), intent(out) :: msq, f
```

```
      real(default), intent(in) :: x
      real(default) :: z, msq0, msq1, tmp
      integer :: type
      type = map%type
      if (s == 0) &
           call msg_fatal (" Applying msq mapping for zero energy")
      select case(type)
      case (NO_MAPPING)
       ⟨Constants for trivial msq mapping⟩
       ⟨Apply trivial msq mapping⟩
      case (S_CHANNEL)
       ⟨Constants for s-channel resonance mapping⟩
       ⟨Apply s-channel resonance mapping⟩
      case (COLLINEAR, INFRARED, RADIATION)
       ⟨Constants for s-channel pole mapping⟩
       ⟨Apply s-channel pole mapping⟩
      case (T_CHANNEL, U_CHANNEL)
       ⟨Constants for t-channel pole mapping⟩
       ⟨Apply t-channel pole mapping⟩
      case default
         call msg_fatal ( " Attempt to apply undefined msq mapping")
      end select
    end subroutine mapping_compute_msq_from_x
```

The inverse mapping

⟨*Mappings: public*⟩+≡

```
    public :: mapping_compute_x_from_msq
```

⟨*Mappings: procedures*⟩+≡

```
    subroutine mapping_compute_x_from_msq (map, s, msq_min, msq_max, msq, f, x)
      type(mapping_t), intent(inout) :: map
      real(default), intent(in) :: s, msq_min, msq_max
      real(default), intent(in) :: msq
      real(default), intent(out) :: f, x
      real(default) :: msq0, msq1, tmp
      integer :: type
      type = map%type
      if (s == 0) &
           call msg_fatal (" Applying inverse msq mapping for zero energy")
  ⟨Modify mapping type if necessary⟩
      select case (type)
      case (NO_MAPPING)
       ⟨Constants for trivial msq mapping⟩
       ⟨Apply inverse trivial msq mapping⟩
      case (S_CHANNEL)
       ⟨Constants for s-channel resonance mapping⟩
       ⟨Apply inverse s-channel resonance mapping⟩
      case (COLLINEAR, INFRARED, RADIATION)
       ⟨Constants for s-channel pole mapping⟩
       ⟨Apply inverse s-channel pole mapping⟩
      case (T_CHANNEL, U_CHANNEL)
       ⟨Constants for t-channel pole mapping⟩
       ⟨Apply inverse t-channel pole mapping⟩
      case default
```

```
        call msg_fatal ( " Attempt to apply undefined msq mapping")
      end select
    end subroutine mapping_compute_x_from_msq
```

Trivial mapping

We simply map the boundaries of the interval $(m_{\min}, m_{\max})$ to $(0, 1)$:

$$m^2 = (1 - x)m_{\min}^2 + xm_{\max}^2; \tag{12.5}$$

the inverse is

$$x = \frac{m^2 - m_{\min}^2}{m_{\max}^2 - m_{\min}^2}. \tag{12.6}$$

Hence

$$f(x) = \frac{m_{\max}^2 - m_{\min}^2}{s}, \tag{12.7}$$

and we have, as required,

$$f(x)\frac{dx}{dm^2} = \frac{1}{s}. \tag{12.8}$$

We store the constant parameters the first time the mapping is called – or, if limits vary, recompute them each time.

⟨*Constants for trivial msq mapping*⟩≡
```
    if (map%variable_limits .or. map%a_unknown) then
      map%a1 = 0
      map%a2 = msq_max - msq_min
      map%a3 = map%a2 / s
      map%a_unknown = .false.
    end if
```

⟨*Apply trivial msq mapping*⟩≡
```
    msq = (1-x) * msq_min + x * msq_max
    f = map%a3
```

⟨*Apply inverse trivial msq mapping*⟩≡
```
    if (map%a2 /= 0) then
      x = (msq - msq_min) / map%a2
    else
      x = 0
    end if
    f = map%a3
```

Breit-Wigner mapping

A Breit-Wigner resonance with mass M and width Γ is flattened by the following mapping:

This mapping does not make much sense if the resonance mass is too low. If this is the case, revert to `NO_MAPPING`. There is a tricky point with this if the mass is too high: `msq_max` is not a constant if structure functions are around. However, switching the type depending on the overall energy does not change the integral, it is just another branching point.

$$m^2 = M(M + t\Gamma), \tag{12.9}$$

where

$$t = \tan \left[(1-x) \arctan \frac{m_{\min}^2 - M^2}{M\Gamma} + x \arctan \frac{m_{\max}^2 - M^2}{M\Gamma} \right]. \qquad (12.10)$$

The inverse:

$$x = \frac{\arctan \frac{m^2 - M^2}{M\Gamma} - \arctan \frac{m_{\min}^2 - M^2}{M\Gamma}}{\arctan \frac{m_{\max}^2 - M^2}{M\Gamma} - \arctan \frac{m_{\min}^2 - M^2}{M\Gamma}} \qquad (12.11)$$

The phase-space factor of this transformation is

$$f(x) = \frac{M\Gamma}{s} \left(\arctan \frac{m_{\max}^2 - M^2}{M\Gamma} - \arctan \frac{m_{\min}^2 - M^2}{M\Gamma} \right) (1 + t^2). \qquad (12.12)$$

This maps any function proportional to

$$g(m^2) = \frac{M\Gamma}{(m^2 - M^2)^2 + M^2\Gamma^2} \qquad (12.13)$$

to a constant times $1/s$.

⟨*Constants for s-channel resonance mapping*⟩≡
```
if (map%variable_limits .or. map%a_unknown) then
   msq0 = map%mass ** 2
   map%a1 = atan ((msq_min - msq0) / (map%mass * map%width))
   map%a2 = atan ((msq_max - msq0) / (map%mass * map%width))
   map%a3 = (map%a2 - map%a1) * (map%mass * map%width) / s
   map%a_unknown = .false.
end if
```

⟨*Apply s-channel resonance mapping*⟩≡
```
z = (1-x) * map%a1 + x * map%a2
if (-pi/2 < z .and. z < pi/2) then
   tmp = tan (z)
   msq = map%mass * (map%mass + map%width * tmp)
   f = map%a3 * (1 + tmp**2)
else
   msq = 0
   f = 0
end if
```

⟨*Apply inverse s-channel resonance mapping*⟩≡
```
tmp = (msq - msq0) / (map%mass * map%width)
x = (atan (tmp) - map%a1) / (map%a2 - map%a1)
f = map%a3 * (1 + tmp**2)
```

Resonance mapping does not make much sense if the resonance mass is outside the kinematical bounds. If this is the case, revert to NO_MAPPING. This is possible even if the kinematical bounds vary from event to event.

⟨*Modify mapping type if necessary*⟩≡
```
if (type == S_CHANNEL) then
   msq0 = map%mass**2
   if (msq0 < msq_min .or. msq0 > msq_max)  type = NO_MAPPING
end if
```

Mapping for massless splittings

This mapping accounts for approximately scale-invariant behavior where $\ln M^2$ is evenly distributed.

$$m^2 = m_{\min}^2 + M^2 \left(\exp(xL) - 1\right) \tag{12.14}$$

where

$$L = \ln \left(\frac{m_{\max}^2 - m_{\min}^2}{M^2} + 1 \right). \tag{12.15}$$

The inverse:

$$x = \frac{1}{L} \ln \left(\frac{m^2 - m_{\min}^2}{M^2} + 1 \right) \tag{12.16}$$

The constant M is a characteristic scale. Above this scale $(m^2 - m_{\min}^2 \gg M^2)$, this mapping behaves like $x \propto \ln m^2$, while below the scale it reverts to a linear mapping.

The phase-space factor is

$$f(x) = \frac{M^2}{s} \exp(xL)\, L. \tag{12.17}$$

A function proportional to

$$g(m^2) = \frac{1}{(m^2 - m_{\min}^2) + M^2} \tag{12.18}$$

is mapped to a constant, i.e., a simple pole near $m_{\min}$ with a regulator mass M.

This type of mapping is useful for massless collinear and infrared singularities, where the scale is stored as the mass parameter. In the radiation case (IR radiation off massive particle), the heavy particle width is the characteristic scale.

⟨*Constants for s-channel pole mapping*⟩≡
```
    if (map%variable_limits .or. map%a_unknown) then
       if (type == RADIATION) then
          msq0 = map%width**2
       else
          msq0 = map%mass**2
       end if
       map%a1 = msq0
       map%a2 = log ((msq_max - msq_min) / msq0 + 1)
       map%a3 = map%a2 / s
       map%a_unknown = .false.
    end if
```

⟨*Apply s-channel pole mapping*⟩≡
```
    msq1 = map%a1 * exp (x * map%a2)
    msq = msq1 - map%a1 + msq_min
    f = map%a3 * msq1
```

⟨*Apply inverse s-channel pole mapping*⟩≡
```
    msq1 = msq - msq_min + map%a1
    x = log (msq1 / map%a1) / map%a2
    f = map%a3 * msq1
```

Mapping for t-channel poles

This is also approximately scale-invariant, and we use the same type of mapping as before. However, we map $1/x$ singularities at both ends of the interval; again, the mapping becomes linear when the distance is less than M^2:

$$m^2 = \begin{cases} m_{\min}^2 + M^2 \left(\exp(xL) - 1 \right) & \text{for } 0 < x < \frac{1}{2} \\ m_{\max}^2 - M^2 \left(\exp((1-x)L) - 1 \right) & \text{for } \frac{1}{2} \le x < 1 \end{cases} \qquad (12.19)$$

where

$$L = 2 \ln \left(\frac{m_{\max}^2 - m_{\min}^2}{2M^2} + 1 \right). \qquad (12.20)$$

The inverse:

$$x = \begin{cases} \frac{1}{L} \ln \left(\frac{m^2 - m_{\min}^2}{M^2} + 1 \right) & \text{for } m^2 < (m_{\max}^2 - m_{\min}^2)/2 \\ 1 - \frac{1}{L} \ln \left(\frac{m_{\max} - m^2}{M^2} + 1 \right) & \text{for } m^2 \ge (m_{\max}^2 - m_{\min}^2)/2 \end{cases} \qquad (12.21)$$

The phase-space factor is

$$f(x) = \begin{cases} \frac{M^2}{s} \exp(xL)\, L. & \text{for } 0 < x < \frac{1}{2} \\ \frac{M^2}{s} \exp((1-x)L)\, L. & \text{for } \frac{1}{2} \le x < 1 \end{cases} \qquad (12.22)$$

A (continuous) function proportional to

$$g(m^2) = \begin{cases} 1/(m^2 - m_{\min}^2) + M^2) & \text{for } m^2 < (m_{\max}^2 - m_{\min}^2)/2 \\ 1/((m_{\max}^2 - m^2) + M^2) & \text{for } m^2 \le (m_{\max}^2 - m_{\min}^2)/2 \end{cases} \qquad (12.23)$$

is mapped to a constant by this mapping, i.e., poles near both ends of the interval.

⟨*Constants for t-channel pole mapping*⟩≡

```
if (map%variable_limits .or. map%a_unknown) then
   msq0 = map%mass**2
   map%a1 = msq0
   map%a2 = 2 * log ((msq_max - msq_min)/(2*msq0) + 1)
   map%a3 = map%a2 / s
   map%a_unknown = .false.
end if
```

⟨*Apply t-channel pole mapping*⟩≡

```
if (x < .5_default) then
   msq1 = map%a1 * exp (x * map%a2)
   msq = msq1 - map%a1 + msq_min
else
   msq1 = map%a1 * exp ((1-x) * map%a2)
   msq = -(msq1 - map%a1) + msq_max
end if
f = map%a3 * msq1
```

⟨*Apply inverse t-channel pole mapping*⟩≡

```
if (msq < (msq_max + msq_min)/2) then
   msq1 = msq - msq_min + map%a1
   x = log (msq1/map%a1) / map%a2
```

```
    else
       msq1 = msq_max - msq + map%a1
       x = 1 - log (msq1/map%a1) / map%a2
    end if
    f = map%a3 * msq1
```

12.1.7 Mappings of the polar angle

The other type of singularity, a simple pole just outside the integration region, can occur in the integration over $\cos\theta$. This applies to exchange of massless (or light) particles.

Double poles (Coulomb scattering) are also possible, but only in certain cases. These are also handled by the single-pole mapping.

The mapping is analogous to the previous m^2 pole mapping, but with a different normalization and notation of variables:

$$\frac{1}{2} \int_{-1}^{1} d\cos\theta \, g(\theta) = \int_{0}^{1} dx \, \frac{d\cos\theta}{dx} \, g(\theta(x)) = \int_{0}^{1} dx \, f(x) \, g(x), \qquad (12.24)$$

where thus

$$f(x) = \frac{1}{2} \frac{d\cos\theta}{dx}. \qquad (12.25)$$

With this mapping, a function of the form

$$g(\theta) = c \frac{dx(\cos\theta)}{d\cos\theta} \qquad (12.26)$$

is mapped to a constant:

$$\int_{-1}^{1} d\cos\theta \, g(\theta) = \int_{0}^{1} dx \, f(x) \, g(\theta(x)) = \int_{0}^{1} dx \, c. \qquad (12.27)$$

⟨*Mappings: public*⟩+≡
```
    public :: mapping_compute_ct_from_x
```

⟨*Mappings: procedures*⟩+≡
```
    subroutine mapping_compute_ct_from_x (map, s, ct, st, f, x)
      type(mapping_t), intent(inout) :: map
      real(default), intent(in) :: s
      real(default), intent(out) :: ct, st, f
      real(default), intent(in) :: x
      real(default) :: tmp, ct1
      select case (map%type)
      case (NO_MAPPING, S_CHANNEL, INFRARED, RADIATION)
        ⟨Apply trivial ct mapping⟩
      case (T_CHANNEL, U_CHANNEL, COLLINEAR)
        ⟨Constants for ct pole mapping⟩
        ⟨Apply ct pole mapping⟩
      case default
        call msg_fatal (" Attempt to apply undefined ct mapping")
      end select
    end subroutine mapping_compute_ct_from_x
```

⟨Mappings: public⟩+≡
```
    public :: mapping_compute_x_from_ct
```

⟨Mappings: procedures⟩+≡
```
    subroutine mapping_compute_x_from_ct (map, s, ct, f, x)
      type(mapping_t), intent(inout) :: map
      real(default), intent(in) :: s
      real(default), intent(in) :: ct
      real(default), intent(out) :: f, x
      real(default) :: ct1
      select case (map%type)
      case (NO_MAPPING, S_CHANNEL, INFRARED, RADIATION)
```
        ⟨Apply inverse trivial ct mapping⟩
```
      case (T_CHANNEL, U_CHANNEL, COLLINEAR)
```
        ⟨Constants for ct pole mapping⟩
        ⟨Apply inverse ct pole mapping⟩
```
      case default
         call msg_fatal (" Attempt to apply undefined inverse ct mapping")
      end select
    end subroutine mapping_compute_x_from_ct
```

Trivial mapping

This is just the mapping of the interval $(-1, 1)$ to $(0, 1)$:

$$\cos\theta = -1 + 2x \qquad (12.28)$$

and

$$f(x) = 1 \qquad (12.29)$$

with the inverse

$$x = \frac{1 + \cos\theta}{2} \qquad (12.30)$$

⟨Apply trivial ct mapping⟩≡
```
    tmp = 2 * (1-x)
    ct = 1 - tmp
    st = sqrt (tmp * (2-tmp))
    f = 1
```

⟨Apply inverse trivial ct mapping⟩≡
```
    x = (ct + 1) / 2
    f = 1
```

Pole mapping

As above for m^2, we simultaneously map poles at both ends of the $\cos\theta$ interval. The formulae are completely analogous:

$$\cos\theta = \begin{cases} \frac{M^2}{s}\left[\exp(xL) - 1\right] - 1 & \text{for } x < \frac{1}{2} \\ -\frac{M^2}{s}\left[\exp((1-x)L) - 1\right] + 1 & \text{for } x \geq \frac{1}{2} \end{cases} \qquad (12.31)$$

where

$$L = 2\ln\frac{M^2 + s}{M^2}. \qquad (12.32)$$

843

Inverse:
$$
x = \begin{cases}
\frac{1}{2L} \ln \frac{1+\cos\theta+M^2/s}{M^2/s} & \text{for } \cos\theta < 0 \\
1 - \frac{1}{2L} \ln \frac{1-\cos\theta+M^2/s}{M^2/s} & \text{for } \cos\theta \geq 0
\end{cases}
\tag{12.33}
$$

The phase-space factor:
$$
f(x) = \begin{cases}
\frac{M^2}{s} \exp(xL)\, L & \text{for } x < \frac{1}{2} \\
\frac{M^2}{s} \exp((1-x)L)\, L & \text{for } x \geq \frac{1}{2}
\end{cases}
\tag{12.34}
$$

⟨*Constants for ct pole mapping*⟩≡
```
if (map%variable_limits .or. map%b_unknown) then
   map%b1 = map%mass**2 / s
   map%b2 = log ((map%b1 + 1) / map%b1)
   map%b3 = 0
   map%b_unknown = .false.
end if
```

⟨*Apply ct pole mapping*⟩≡
```
if (x < .5_default) then
   ct1 = map%b1 * exp (2 * x * map%b2)
   ct = ct1 - map%b1 - 1
else
   ct1 = map%b1 * exp (2 * (1-x) * map%b2)
   ct = -(ct1 - map%b1) + 1
end if
if (ct >= -1 .and. ct <= 1) then
   st = sqrt (1 - ct**2)
   f = ct1 * map%b2
else
   ct = 1;   st = 0;   f = 0
end if
```

⟨*Apply inverse ct pole mapping*⟩≡
```
if (ct < 0) then
   ct1 = ct + map%b1 + 1
   x = log (ct1 / map%b1) / (2 * map%b2)
else
   ct1 = -ct + map%b1 + 1
   x = 1 - log (ct1 / map%b1) / (2 * map%b2)
end if
f = ct1 * map%b2
```

12.2   Phase-space trees

The phase space evaluation is organized in terms of trees, where each branch corresponds to three integrations: m^2, $\cos\theta$, and ϕ. The complete tree thus makes up a specific parameterization of the multidimensional phase-space integral. For the multi-channel integration, the phase-space tree is a single channel.

The trees imply mappings of formal Feynman tree graphs into arrays of integer numbers: Each branch, corresponding to a particular line in the graph, is assigned an integer code c (with kind value TC = tree code).

In this integer, each bit determines whether a particular external momentum flows through the line. The external branches therefore have codes $1, 2, 4, 8, \ldots$. An internal branch has those bits ORed corresponding to the momenta flowing through it. For example, a branch with momentum $p_1 + p_4$ has code $2^0 + 2^3 = 1 + 8 = 9$.

There is a two-fold ambiguity: Momentum conservation implies that the branch with code

$$c_0 = \sum_{i=1}^{n(\text{ext})} 2^{i-1} \tag{12.35}$$

i.e. the branch with momentum $p_1 + p_2 + \ldots p_n$ has momentum zero, which is equivalent to tree code 0 by definition. Correspondingly,

$$c \quad \text{and} \quad c_0 - c = c \text{ XOR } c_0 \tag{12.36}$$

are equivalent. E.g., if there are five externals with codes $c = 1, 2, 4, 8, 16$, then $c = 9$ and $\bar{c} = 31 - 9 = 22$ are equivalent.

This ambiguity may be used to assign a direction to the line: If all momenta are understood as outgoing, $c = 9$ in the example above means $p_1 + p_4$, but $c = 22$ means $p_2 + p_3 + p_5 = -(p_1 + p_4)$.

Here we make use of the ambiguity in a slightly different way. First, the initial particles are singled out as those externals with the highest bits, the IN-bits. (Here: 8 and 16 for a $2 \to 3$ scattering process, 16 only for a $1 \to 4$ decay.) Then we invert those codes where all IN-bits are set. For a decay process this maps each tree of an equivalence class onto a unique representative (that one with the smallest integer codes). For a scattering process we proceed further:

The ambiguity remains in all branches where only one IN-bit is set, including the initial particles. If there are only externals with this property, we have an s-channel graph which we leave as it is. In all other cases, an internal with only one IN-bit is a t-channel line, which for phase space integration should be associated with one of the initial momenta as a reference axis. We take that one whose bit is set in the current tree code. (E.g., for branch $c = 9$ we use the initial particle $c = 8$ as reference axis, whereas for the same branch we would take $c = 16$ if it had been assigned $\bar{c} = 31 - 9 = 22$ as tree code.) Thus, different ways of coding the same t-channel graph imply different phase space parameterizations.

s-channel graphs have a unique parameterization. The same sets of parameterizations are used for t-channel graphs, except for the reference frames of their angular parts. We map each t-channel graph onto an s-channel graph as follows:

Working in ascending order, for each t-channel line (whose code has exactly one IN-bit set) the attached initial line is flipped upstream, while the outgoing

line is flipped downstream. (This works only if *t*-channel graphs are always parameterized beginning at their outer vertices, which we require as a restriction.) After all possible flips have been applied, we have an *s*-channel graph. We only have to remember the initial particle a vertex was originally attached to.

⟨phs_trees.f90⟩≡
  ⟨*File header*⟩

  module phs_trees

  ⟨*Use kinds*⟩
    use kinds, only: TC !NODEP!
  ⟨*Use strings*⟩
    use constants, only: twopi, twopi2, twopi5 !NODEP!
  ⟨*Use file utils*⟩
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use permutations, only: permutation_t, permutation_size
    use permutations, only: permutation_init, permutation_find
    use permutations, only: tc_decay_level, tc_permute
    use models
    use flavors
    use mappings

  ⟨*Standard module head*⟩

  ⟨*PHS trees: public*⟩

  ⟨*PHS trees: types*⟩

  contains

  ⟨*PHS trees: procedures*⟩

  end module phs_trees

12.2.1 Particles

We define a particle type which contains only four-momentum and invariant mass squared, and a flag that tells whether the momentum is filled or not.

⟨*PHS trees: public*⟩≡
    public :: phs_prt_t

⟨*PHS trees: types*⟩≡
    type :: phs_prt_t
       private
       logical :: defined = .false.
       type(vector4_t) :: p
       real(default) :: p2
    end type phs_prt_t

Set contents:

⟨*PHS trees: public*⟩+≡
    public :: phs_prt_set_defined

```
    public :: phs_prt_set_undefined
    public :: phs_prt_set_momentum
    public :: phs_prt_set_msq
```

⟨*PHS trees: procedures*⟩≡
```
    elemental subroutine phs_prt_set_defined (prt)
      type(phs_prt_t), intent(inout) :: prt
      prt%defined = .true.
    end subroutine phs_prt_set_defined

    elemental subroutine phs_prt_set_undefined (prt)
      type(phs_prt_t), intent(inout) :: prt
      prt%defined = .false.
    end subroutine phs_prt_set_undefined

    elemental subroutine phs_prt_set_momentum (prt, p)
      type(phs_prt_t), intent(inout) :: prt
      type(vector4_t), intent(in) :: p
      prt%p = p
    end subroutine phs_prt_set_momentum

    elemental subroutine phs_prt_set_msq (prt, p2)
      type(phs_prt_t), intent(inout) :: prt
      real(default), intent(in) :: p2
      prt%p2 = p2
    end subroutine phs_prt_set_msq
```

Access methods:

⟨*PHS trees: public*⟩+≡
```
    public :: phs_prt_is_defined
    public :: phs_prt_get_momentum
    public :: phs_prt_get_msq
```

⟨*PHS trees: procedures*⟩+≡
```
    elemental function phs_prt_is_defined (prt) result (defined)
      logical :: defined
      type(phs_prt_t), intent(in) :: prt
      defined = prt%defined
    end function phs_prt_is_defined

    elemental function phs_prt_get_momentum (prt) result (p)
      type(vector4_t) :: p
      type(phs_prt_t), intent(in) :: prt
      p = prt%p
    end function phs_prt_get_momentum

    elemental function phs_prt_get_msq (prt) result (p2)
      real(default) :: p2
      type(phs_prt_t), intent(in) :: prt
      p2 = prt%p2
    end function phs_prt_get_msq
```

Addition of momenta (invariant mass square is computed).

⟨*PHS trees: public*⟩+≡

```
    public :: phs_prt_combine
```

⟨*PHS trees: procedures*⟩+≡
```
    elemental subroutine phs_prt_combine (prt, prt1, prt2)
      type(phs_prt_t), intent(inout) :: prt
      type(phs_prt_t), intent(in) :: prt1, prt2
      prt%defined = .true.
      prt%p = prt1%p + prt2%p
      prt%p2 = prt%p ** 2
    end subroutine phs_prt_combine
```

Output

⟨*PHS trees: public*⟩+≡
```
    public :: phs_prt_write
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_prt_write (prt, unit)
      type(phs_prt_t), intent(in) :: prt
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      if (prt%defined) then
         call vector4_write (prt%p, u)
         write (u, *) "M2 =", prt%p2
      else
         write (u, *) "[undefined]"
      end if
    end subroutine phs_prt_write
```

12.2.2   The phase-space tree type

Definition

In the concrete implementation, each branch c may have two *daughters* c_1 and c_2 such that $c_1 + c_2 = c$, a *sibling* c_s and a *mother* c_m such that $c + c_s = c_m$, and a *friend* which is kept during flips, such that it can indicate a fixed reference frame. Absent entries are set $c = 0$.

First, declare the branch type. There is some need to have this public. Give initializations for all components, so no `init` routine is necessary. The branch has some information about the associated coordinates and about connections.

⟨*PHS trees: types*⟩+≡
```
    type :: phs_branch_t
      private
      logical :: set = .false.
      logical :: inverted_decay = .false.
      logical :: inverted_axis = .false.
      integer(TC) :: mother = 0
      integer(TC) :: sibling = 0
      integer(TC) :: friend = 0
      integer(TC) :: origin = 0
      integer(TC), dimension(2) :: daughter = 0
      integer :: firstborn = 0
```

```
      logical :: has_children = .false.
      logical :: has_friend = .false.
   end type phs_branch_t
```

The tree type: No initialization, this is done by `phs_tree_init`. In addition to the branch array which

The branches are collected in an array which holds all possible branches, of which only a few are set. After flips have been applied, the branch $c_M = \sum_{i=1}^{n(\text{fin})} 2^{i-1}$ must be there, indicating the mother of all decay products. In addition, we should check for consistency at the beginning.

`n_branches` is the number of those actually set. `n_externals` defines the number of significant bit, and `mask` is a code where all bits are set. Analogous: `n_in` and `mask_in` for the incoming particles.

The `mapping` array contains the mappings associated to the branches (corresponding indices). The array `mass_sum` contains the sum of the real masses of the external final-state particles associated to the branch. During phase-space evaluation, this determines the boundaries.

⟨*PHS trees: public*⟩+≡
```
   public :: phs_tree_t
```

⟨*PHS trees: types*⟩+≡
```
   type :: phs_tree_t
      private
      integer :: n_branches, n_externals, n_in, n_msq, n_angles
      integer(TC) :: n_branches_tot, n_branches_out
      integer(TC) :: mask, mask_in, mask_out
      type(phs_branch_t), dimension(:), allocatable :: branch
      type(mapping_t), dimension(:), allocatable :: mapping
      real(default), dimension(:), allocatable :: mass_sum
   end type phs_tree_t
```

The maximum number of external particles that can be represented is related to the bit size of the integer that stores binary codes. With the default integer of 32 bit on common machines, this is more than enough space. If `TC` is actually the default integer kind, there is no need to keep it separate, but doing so marks this as a special type of integer. So, just state that the maximum number is 32:

⟨*Limits: public parameters*⟩+≡
```
   integer, parameter, public :: MAX_EXTERNAL = 32
```

Constructor and destructor

Allocate memory for a phase-space tree with given number of externals and incoming. The number of allocated branches can easily become large, but appears manageable for realistic cases, e.g., for `n_in=2` and `n_out=8` we get $2^{10} - 1 = 1023$.

⟨*PHS trees: public*⟩+≡
```
   public :: phs_tree_init
   public :: phs_tree_final
```

Here we set the masks for incoming and for all externals.

⟨*PHS trees: procedures*⟩+≡
```
elemental subroutine phs_tree_init (tree, n_in, n_out, n_masses, n_angles)
  type(phs_tree_t), intent(inout) :: tree
  integer, intent(in) :: n_in, n_out, n_masses, n_angles
  integer(TC) :: i
  tree%n_externals = n_in + n_out
  tree%n_branches_tot = 2**(n_in+n_out) - 1
  tree%n_branches_out = 2**n_out - 1
  tree%mask = 0
  do i = 0, n_in + n_out - 1
     tree%mask = ibset (tree%mask, i)
  end do
  tree%n_in = n_in
  tree%mask_in = 0
  do i = n_out, n_in + n_out - 1
     tree%mask_in = ibset (tree%mask_in, i)
  end do
  tree%mask_out = ieor (tree%mask, tree%mask_in)
  tree%n_msq = n_masses
  tree%n_angles = n_angles
  allocate (tree%branch (tree%n_branches_tot))
  tree%n_branches  = 0
  allocate (tree%mapping (tree%n_branches_out))
  allocate (tree%mass_sum (tree%n_branches_out))
end subroutine phs_tree_init

elemental subroutine phs_tree_final (tree)
  type(phs_tree_t), intent(inout) :: tree
  deallocate (tree%branch)
  deallocate (tree%mapping)
  deallocate (tree%mass_sum)
end subroutine phs_tree_final
```

Screen output

Write only the branches that are set:

⟨*PHS trees: public*⟩+≡
```
public :: phs_tree_write
```

⟨*PHS trees: procedures*⟩+≡
```
subroutine phs_tree_write (tree, unit)
  type(phs_tree_t), intent(in) :: tree
  integer, intent(in), optional :: unit
  integer :: u
  integer(TC) :: k
  u = output_unit (unit);  if (u < 0)  return
  write (u,'(1X,A,I2,5X,A,I3)') &
       'External:', tree%n_externals, 'Mask:', tree%mask
  write (u,'(1X,A,I2,5X,A,I3)') &
       'Incoming:', tree%n_in, 'Mask:', tree%mask_in
  write (u,'(1X,A,I2,5X,A,I3)') &
       'Branches:', tree%n_branches
```

```
      do k = size (tree%branch), 1, -1
         if (tree%branch(k)%set) &
              call phs_branch_write (tree%branch(k), unit=unit, kval=k)
      end do
      do k = 1, size (tree%mapping)
         call mapping_write (tree%mapping (k), unit)
      end do
      do k = 1, size (tree%mass_sum)
         if (tree%branch(k)%set) then
            write (u, *) k, "mass_sum =", tree%mass_sum(k)
         end if
      end do
  end subroutine phs_tree_write

  subroutine phs_branch_write (b, unit, kval)
    type(phs_branch_t), intent(in) :: b
    integer, intent(in), optional :: unit
    integer(TC), intent(in), optional :: kval
    integer :: u
    integer(TC) :: k
    character(len=6) :: tmp
    character(len=1) :: firstborn(2), sign_decay, sign_axis
    integer :: i
    u = output_unit (unit);  if (u < 0)  return
    k = 0;  if (present (kval))  k = kval
    if (b%origin /= 0) then
       write(tmp, '(A,I4,A)') '(', b%origin, ')'
    else
       tmp = ' '
    end if
    do i=1, 2
       if (b%firstborn == i) then
          firstborn(i) = "*"
       else
          firstborn(i) = " "
       end if
    end do
    if (b%inverted_decay) then
       sign_decay = "-"
    else
       sign_decay = "+"
    end if
    if (b%inverted_axis) then
       sign_axis = "-"
    else
       sign_axis = "+"
    end if
    if (b%has_children) then
       if (b%has_friend) then
          write(u,'(1X,A,I4,1x,A,2X,A,I4,A,I4,A,2X,A,3X,A,I4)') &
               &    '*', k, tmp, &
               &    'Daughters: ', &
               &    b%daughter(1), firstborn(1), &
               &    b%daughter(2), firstborn(2), sign_decay, &
```

```
              &   'Friend:    ', b%friend
        else
           write(u,'(1X,A,I4,1x,A,2X,A,I4,A,I4,A,2X,A,2X,A)') &
                 &   '*', k, tmp, &
                 &   'Daughters: ', &
                 &   b%daughter(1), firstborn(1), &
                 &   b%daughter(2), firstborn(2), sign_decay, &
                 &   '(axis '//sign_axis//')'
        end if
     else
        write(u,'(2X,I4,3X,A,I4,I4)') k
     end if
  end subroutine phs_branch_write
```

12.2.3 PHS tree setup

Transformation into an array of branch codes and back

Assume that the tree/array has been created before with the appropriate length
and is empty.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_from_array
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_from_array (tree, a)
      type(phs_tree_t), intent(inout) :: tree
      integer(TC), dimension(:), intent(in) :: a
      integer :: i
      integer(TC) :: k
```
⟨*Set branches from array* a⟩
⟨*Set external branches if necessary*⟩
⟨*Check number of branches*⟩
⟨*Determine the connections*⟩
```
    contains
```
⟨*Subroutine: set relatives*⟩
```
    end subroutine phs_tree_from_array
```

First, set all branches specified by the user. If all IN-bits are set, we invert the
branch code.

⟨*Set branches from array* a⟩≡
```
    do i=1, size(a)
       k = a(i)
       if (iand(k, tree%mask_in) == tree%mask_in)  k = ieor(tree%mask, k)
       tree%branch(k)%set = .true.
       tree%n_branches = tree%n_branches+1
    end do
```

The external branches are understood, so set them now if not yet done. In all
cases ensure that the representative with one bit set is used, except for decays
where the in-particle is represented by all OUT-bits set instead.

⟨*Set external branches if necessary*⟩≡
```
    do i=0, tree%n_externals-1
```

852

```
        k = ibset(0,i)
        if (iand(k, tree%mask_in) == tree%mask_in)  k = ieor(tree%mask, k)
        if (tree%branch(ieor(tree%mask, k))%set) then
           tree%branch(ieor(tree%mask, k))%set = .false.
           tree%branch(k)%set = .true.
        else if (.not.tree%branch(k)%set) then
           tree%branch(k)%set = .true.
           tree%n_branches = tree%n_branches+1
        end if
     end do
```

Now the number of branches set can be checked. Here we assume that the tree is binary. For three externals there are three branches in total, and for each additional external branch we get another internal one.

⟨*Check number of branches*⟩≡
```
     if (tree%n_branches /= tree%n_externals*2-3) then
        call phs_tree_write (tree)
        call msg_bug &
             & (" Wrong number of branches set in phase space tree")
     end if
```

For all branches that are set, except for the externals, we try to find the daughter branches:

⟨*Determine the connections*⟩≡
```
     do k=1, size (tree%branch)
        if (tree%branch(k)%set .and. tc_decay_level (k) /= 1) then
           call branch_set_relatives(k)
        end if
     end do
```

To this end, we scan all codes less than the current code, whether we can find two branches which are set and which together give the current code. After that, the tree may still not be connected, but at least we know if a branch does not have daughters: This indicates some inconsistency.

The algorithm ensures that, at this stage, the first daughter has a smaller code value than the second one.

⟨*Subroutine: set relatives*⟩≡
```
     subroutine branch_set_relatives (k)
       integer(TC), intent(in) :: k
       integer(TC) :: m,n
       do m=1, k-1
          if(iand(k,m)==m) then
             n = ieor(k,m)
             if ( tree%branch(m)%set .and. tree%branch(n)%set ) then
                tree%branch(k)%daughter(1) = m;  tree%branch(k)%daughter(2) = n
                tree%branch(m)%mother      = k;  tree%branch(n)%mother      = k
                tree%branch(m)%sibling     = n;  tree%branch(n)%sibling     = m
                tree%branch(k)%has_children = .true.
                return
             end if
          end if
       end do
       call phs_tree_write (tree)
       call msg_bug &
            & (" Missing daughter branch(es) in phase space tree")
```

```
            end subroutine branch_set_relatives
```

The inverse: this is trivial, fortunately.

Flip _t_-channel into _s_-channel

Flipping the tree is done upwards, beginning from the decay products. First we
select a _t_-channel branch `k`: one which is set, which does have an IN-bit, and
which is not an external particle.

Next, we determine the adjacent in-particle (called the 'friend' `f` here, since
it will provide the reference axis for the angular integration). In addition, we
look for the 'mother' and 'sibling' of this particle. If the latter field is empty, we
select the (unique) other out-particle which has no mother, calling the internal
subroutine `find_orphan`.

The flip is done as follows: We assume that the first daughter `d` is an _s_-
channel line, which is true if the daughters are sorted. This will stay the first
daughter. The second one is a _t_-channel line; it is exchanged with the 'sibling'
`s`. The new line which replaces the branch `k` is just the sum of `s` and `d`. In
addition, we have to rearrange the relatives of `s` and `d`, as well of `f`.

Finally, we flip 'sibling' and 'friend' and set the new _s_-channel branch `n`
which replaces the _t_-channel branch `k`. After this is complete, we are ready to
execute another flip.

[Although the friend is not needed for the final flip, since it would be an
initial particle anyway, we need to know whether we have _t_- or _u_-channel.]

⟨_PHS trees: public_⟩+≡
```
    public :: phs_tree_flip_t_to_s_channel
```

⟨_PHS trees: procedures_⟩+≡
```
    subroutine phs_tree_flip_t_to_s_channel (tree)
      type(phs_tree_t), intent(inout) :: tree
      integer(TC) :: k, f, m, n, d, s
      if (tree%n_in == 2) then
         FLIP: do k=3, tree%mask-1
            if (.not. tree%branch(k)%set) cycle FLIP
            f = iand(k,tree%mask_in)
            if (f==0 .or. f==k) cycle FLIP
            m = tree%branch(k)%mother
            s = tree%branch(k)%sibling
            if (s==0) call find_orphan(s)
            d = tree%branch(k)%daughter(1)
            n = ior(d,s)
            tree%branch(k)%set = .false.
            tree%branch(n)%set = .true.
            tree%branch(n)%origin = k
            tree%branch(n)%daughter(1) = d; tree%branch(d)%mother  = n
            tree%branch(n)%daughter(2) = s; tree%branch(s)%mother  = n
            tree%branch(n)%has_children = .true.
            tree%branch(d)%sibling = s;  tree%branch(s)%sibling = d
            tree%branch(n)%sibling = f;  tree%branch(f)%sibling = n
            tree%branch(n)%mother      = m
            tree%branch(f)%mother      = m
            if (m/=0) then
```

```
                tree%branch(m)%daughter(1) = n
                tree%branch(m)%daughter(2) = f
             end if
             tree%branch(n)%friend = f
             tree%branch(n)%has_friend = .true.
             tree%branch(n)%firstborn = 2
          end do FLIP
       end if
     contains
       subroutine find_orphan(s)
         integer(TC) :: s
         do s=1, tree%mask_out
            if (tree%branch(s)%set .and. tree%branch(s)%mother==0) return
         end do
         call phs_tree_write (tree)
         call msg_bug (" Can't flip phase space tree to channel")
       end subroutine find_orphan
     end subroutine phs_tree_flip_t_to_s_channel
```

After the tree has been flipped, one may need to determine what has become
of a particular t-channel branch. This function gives the bincode of the flipped
tree. If the original bincode does not contain IN-bits, we leave it as it is.

⟨*PHS trees: procedures*⟩+≡
```
     function tc_flipped (tree, kt) result (ks)
       type(phs_tree_t), intent(in) :: tree
       integer(TC), intent(in) :: kt
       integer(TC) :: ks
       if (iand (kt, tree%mask_in) == 0) then
          ks = kt
       else
          ks = tree%branch(iand (kt, tree%mask_out))%mother
       end if
     end function tc_flipped
```

Scan a tree and make sure that the first daughter has always a smaller code
than the second one. Furthermore, delete any `friend` entry in the root branch
– this branching has the incoming particle direction as axis anyway. Keep track
of reordering by updating `inverted_axis`, `inverted_decay` and `firstborn`.

⟨*PHS trees: public*⟩+≡
```
     public :: phs_tree_canonicalize
```

⟨*PHS trees: procedures*⟩+≡
```
     subroutine phs_tree_canonicalize (tree)
       type(phs_tree_t), intent(inout) :: tree
       integer :: n_out
       integer(TC) :: k_out
       call branch_canonicalize (tree%branch(tree%mask_out))
       n_out = tree%n_externals - tree%n_in
       k_out = tree%mask_out
       if (tree%branch(k_out)%has_friend &
            & .and. tree%branch(k_out)%friend == ibset (0, n_out)) then
          tree%branch(k_out)%inverted_axis = .not.tree%branch(k_out)%inverted_axis
       end if
```

```
      tree%branch(k_out)%has_friend = .false.
      tree%branch(k_out)%friend = 0
   contains
     recursive subroutine branch_canonicalize (b)
       type(phs_branch_t), intent(inout) :: b
       integer(TC) :: d1, d2
       if (b%has_children) then
          d1 = b%daughter(1)
          d2 = b%daughter(2)
          if (d1 > d2) then
             b%daughter(1) = d2
             b%daughter(2) = d1
             b%inverted_decay = .not.b%inverted_decay
             if (b%firstborn /= 0)  b%firstborn = 3 - b%firstborn
          end if
          call branch_canonicalize (tree%branch(b%daughter(1)))
          call branch_canonicalize (tree%branch(b%daughter(2)))
       end if
     end subroutine branch_canonicalize
   end subroutine phs_tree_canonicalize
```

Mappings

Initialize a mapping for the current tree. This is done while reading from file,
so the mapping parameters are read, but applied to the flipped tree. Thus, the
size of the array of mappings is given by the number of outgoing particles only.

⟨*PHS trees: public*⟩+≡

```
    public :: phs_tree_init_mapping
```

⟨*PHS trees: procedures*⟩+≡

```
    subroutine phs_tree_init_mapping (tree, k, type, pdg, model)
      type(phs_tree_t), intent(inout) :: tree
      integer(TC), intent(in) :: k
      type(string_t), intent(in) :: type
      integer, intent(in) :: pdg
      type(model_t), intent(in), target :: model
      integer(TC) :: kk
      kk = tc_flipped (tree, k)
      call mapping_init (tree%mapping(kk), kk, type, pdg, model)
    end subroutine phs_tree_init_mapping
```

Set the physical parameters for the mapping, using a specific parameter set.
Also set the mass sum array.

⟨*PHS trees: public*⟩+≡

```
    public :: phs_tree_set_mapping_parameters
```

⟨*PHS trees: procedures*⟩+≡

```
    subroutine phs_tree_set_mapping_parameters &
        (tree, mapping_defaults, variable_limits)
      type(phs_tree_t), intent(inout) :: tree
      type(mapping_defaults_t), intent(in) :: mapping_defaults
      logical, intent(in) :: variable_limits
      integer(TC) :: k
```

```
      do k = 1, tree%n_branches_out
         call mapping_set_parameters &
              (tree%mapping(k), mapping_defaults, variable_limits)
      end do
    end subroutine phs_tree_set_mapping_parameters
```

Kinematics

Fill the mass sum array, starting from the external particles and working down to the tree root. For each bincode k we scan the bits in k; if only one is set, we take the physical mass of the corresponding external particle; if more then one is set, we sum up the two masses (which we know have already been set).

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_set_mass_sum
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_set_mass_sum (tree, flv)
      type(phs_tree_t), intent(inout) :: tree
      type(flavor_t), dimension(:), intent(in) :: flv
      integer(TC) :: k
      integer :: i
      tree%mass_sum = 0
      do k = 1, tree%n_branches_out
         do i = 0, size (flv) - 1
            if (btest(k,i)) then
               if (ibclr(k,i) == 0) then
                  tree%mass_sum(k) = flavor_get_mass (flv(i+1))
               else
                  tree%mass_sum(k) = &
                       tree%mass_sum(ibclr(k,i)) + tree%mass_sum(ibset(0,i))
               end if
               exit
            end if
         end do
      end do
    end subroutine phs_tree_set_mass_sum
```

Structural comparison

This function allows to check whether one tree is the permutation of another one. The permutation is applied to the second tree in the argument list. We do not make up a temporary permuted tree, but compare the two trees directly. The branches are scanned recursively, where for each daughter we check the friend and the mapping as well. Once a discrepancy is found, the recursion is exited immediately.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_equivalent
```

⟨*PHS trees: procedures*⟩+≡
```
    function phs_tree_equivalent (t1, t2, perm) result (is_equal)
      type(phs_tree_t), intent(in) :: t1, t2
```

```
      type(permutation_t), intent(in) :: perm
      logical :: equal, is_equal
      integer(TC) :: k1, k2, mask_in
      k1 = t1%mask_out
      k2 = t2%mask_out
      mask_in = t1%mask_in
      equal = .true.
      call check (t1%branch(k1), t2%branch(k2), k1, k2)
      is_equal = equal
    contains
      recursive subroutine check (b1, b2, k1, k2)
        type(phs_branch_t), intent(in) :: b1, b2
        integer(TC), intent(in) :: k1, k2
        integer(TC), dimension(2) :: d1, d2, pd2
        integer :: i
        if (.not.b1%has_friend .and. .not.b2%has_friend) then
            equal = .true.
        else if (b1%has_friend .and. b2%has_friend) then
            equal = (b1%friend == tc_permute (b2%friend, perm, mask_in))
        end if
        if (equal) then
            if (b1%has_children .and. b2%has_children) then
                d1 = b1%daughter
                d2 = b2%daughter
                do i=1, 2
                    pd2(i) = tc_permute (d2(i), perm, mask_in)
                end do
                if (d1(1)==pd2(1) .and. d1(2)==pd2(2)) then
                    equal = (b1%firstborn == b2%firstborn)
                    if (equal) call check &
                        &    (t1%branch(d1(1)), t2%branch(d2(1)), d1(1), d2(1))
                    if (equal) call check &
                        &    (t1%branch(d1(2)), t2%branch(d2(2)), d1(2), d2(2))
                else if (d1(1)==pd2(2) .and. d1(2)==pd2(1)) then
                    equal = ( (b1%firstborn == 0 .and. b2%firstborn == 0) &
                        &   .or. (b1%firstborn == 3 - b2%firstborn) )
                    if (equal) call check &
                        &    (t1%branch(d1(1)), t2%branch(d2(2)), d1(1), d2(2))
                    if (equal) call check &
                        &    (t1%branch(d1(2)), t2%branch(d2(1)), d1(2), d2(1))
                else
                    equal = .false.
                end if
            end if
        end if
        if (equal) then
            equal = (t1%mapping(k1) == t2%mapping(k2))
        end if
      end subroutine check
  end function phs_tree_equivalent
```

Scan two decay trees and determine the correspondence of mass variables, i.e.,
the permutation that transfers the ordered list of mass variables belonging to

the second tree into the first one. Mass variables are assigned beginning from
branches and ending at the root.

⟨*PHS trees: public*⟩+≡

```
    public :: phs_tree_find_msq_permutation
```

⟨*PHS trees: procedures*⟩+≡

```
    subroutine phs_tree_find_msq_permutation (tree1, tree2, perm2, msq_perm)
      type(phs_tree_t), intent(in) :: tree1, tree2
      type(permutation_t), intent(in) :: perm2
      type(permutation_t), intent(out) :: msq_perm
      type(permutation_t) :: perm1
      integer(TC) :: mask_in, root
      integer(TC), dimension(:), allocatable :: index1, index2
      integer :: i
      allocate (index1 (tree1%n_msq), index2 (tree2%n_msq))
      call permutation_init (perm1, permutation_size (perm2))
      mask_in = tree1%mask_in
      root = tree1%mask_out
      i = 0
      call tree_scan (tree1, root, perm1, index1)
      i = 0
      call tree_scan (tree2, root, perm2, index2)
      call permutation_find (msq_perm, index1, index2)
    contains
      recursive subroutine tree_scan (tree, k, perm, index)
        type(phs_tree_t), intent(in) :: tree
        integer(TC), intent(in) :: k
        type(permutation_t), intent(in) :: perm
        integer, dimension(:), intent(inout) :: index
        if (tree%branch(k)%has_children) then
            call tree_scan (tree, tree%branch(k)%daughter(1), perm, index)
            call tree_scan (tree, tree%branch(k)%daughter(2), perm, index)
            i = i + 1
            if (i <= size (index))  index(i) = tc_permute (k, perm, mask_in)
        end if
      end subroutine tree_scan
    end subroutine phs_tree_find_msq_permutation
```

⟨*PHS trees: public*⟩+≡

```
    public :: phs_tree_find_angle_permutation
```

⟨*PHS trees: procedures*⟩+≡

```
    subroutine phs_tree_find_angle_permutation &
        (tree1, tree2, perm2, angle_perm, sig2)
      type(phs_tree_t), intent(in) :: tree1, tree2
      type(permutation_t), intent(in) :: perm2
      type(permutation_t), intent(out) :: angle_perm
      logical, dimension(:), allocatable, intent(out) :: sig2
      type(permutation_t) :: perm1
      integer(TC) :: mask_in, root
      integer(TC), dimension(:), allocatable :: index1, index2
      logical, dimension(:), allocatable :: sig1
      integer :: i
      allocate (index1 (tree1%n_angles), index2 (tree2%n_angles))
      allocate (sig1 (tree1%n_angles), sig2 (tree2%n_angles))
```

```
      call permutation_init (perm1, permutation_size (perm2))
      mask_in = tree1%mask_in
      root = tree1%mask_out
      i = 0
      call tree_scan (tree1, root, perm1, index1, sig1)
      i = 0
      call tree_scan (tree2, root, perm2, index2, sig2)
      call permutation_find (angle_perm, index1, index2)
    contains
      recursive subroutine tree_scan (tree, k, perm, index, sig)
        type(phs_tree_t), intent(in) :: tree
        integer(TC), intent(in) :: k
        type(permutation_t), intent(in) :: perm
        integer, dimension(:), intent(inout) :: index
        logical, dimension(:), intent(inout) :: sig
        integer(TC) :: k1, k2, kp
        logical :: s
        if (tree%branch(k)%has_children) then
          k1 = tree%branch(k)%daughter(1)
          k2 = tree%branch(k)%daughter(2)
          s = (tc_permute(k1, perm, mask_in) < tc_permute(k2, perm, mask_in))
          kp = tc_permute (k, perm, mask_in)
          i = i + 1
          index(i) = kp
          sig(i) = s
          i = i + 1
          index(i) = - kp
          sig(i) = s
          call tree_scan (tree, k1, perm, index, sig)
          call tree_scan (tree, k2, perm, index, sig)
        end if
      end subroutine tree_scan
    end subroutine phs_tree_find_angle_permutation
```

12.2.4   Phase-space evaluation

Determine momenta

This is done in two steps: First the masses are determined. This step may
fail, in which case `ok` is set to false. If successful, we generate angles and
the actual momenta. The array `decay_p` serves for transferring the individual
three-momenta of the daughter particles in their mother rest frame from the
mass generation to the momentum generation step.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_compute_momenta_from_x
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_compute_momenta_from_x &
        (tree, prt, factor, volume, sqrts, x, ok)
      type(phs_tree_t), intent(inout) :: tree
      type(phs_prt_t), dimension(:), intent(inout) :: prt
      real(default), intent(out) :: factor, volume
      real(default), intent(in) :: sqrts
```

```
      real(default), dimension(:), intent(in) :: x
      logical, intent(out) :: ok
      real(default), dimension(tree%mask_out) :: decay_p
      integer :: n1, n2
      n1 = tree%n_msq
      n2 = n1 + tree%n_angles
      call phs_tree_set_msq &
           (tree, prt, factor, volume, decay_p, sqrts, x(1:n1), ok)
      if (ok) call phs_tree_set_angles &
           (tree, prt, factor, decay_p, sqrts, x(n1+1:n2))
    end subroutine phs_tree_compute_momenta_from_x
```

Mass generation is done recursively. The `ok` flag causes the filled tree to be discarded if set to `.false.`. This happens if a three-momentum turns out to be imaginary, indicating impossible kinematics. The index `ix` tells us how far we have used up the input array `x`.

⟨*PHS trees: procedures*⟩+≡

```
    subroutine phs_tree_set_msq &
         (tree, prt, factor, volume, decay_p, sqrts, x, ok)
      type(phs_tree_t), intent(inout) :: tree
      type(phs_prt_t), dimension(:), intent(inout) :: prt
      real(default), intent(out) :: factor, volume
      real(default), dimension(:), intent(out) :: decay_p
      real(default), intent(in) :: sqrts
      real(default), dimension(:), intent(in) :: x
      logical, intent(out) :: ok
      integer :: ix
      integer(TC) :: k
      real(default) :: m_tot
      ok =.true.
      ix = 1
      k  = tree%mask_out
      m_tot = tree%mass_sum(k)
      decay_p(k) = 0.
      if (m_tot < sqrts .or. k == 1) then
         if (tree%branch(k)%has_children) then
            call set_msq_x (tree%branch(k), k, factor, volume, .true.)
         else
            factor = 1
            volume = 1
         end if
      else
         ok = .false.
      end if
    contains
      recursive subroutine set_msq_x (b, k, factor, volume, initial)
        type(phs_branch_t), intent(in) :: b
        integer(TC), intent(in) :: k
        real(default), intent(out) :: factor, volume
        logical, intent(in) :: initial
        real(default) :: msq, m, m_min, m_max, m1, m2, msq1, msq2, lda, rlda
        integer(TC) :: k1, k2
        real(default) :: f1, f2, v1, v2
```

861

```
        k1 = b%daughter(1);   k2 = b%daughter(2)
        if (tree%branch(k1)%has_children) then
            call set_msq_x (tree%branch(k1), k1, f1, v1, .false.)
            if (.not.ok) return
        else
            f1 = 1;   v1 = 1
        end if
        if (tree%branch(k2)%has_children) then
            call set_msq_x (tree%branch(k2), k2, f2, v2, .false.)
            if (.not.ok) return
        else
            f2 = 1;   v2 = 1
        end if
        m_min = tree%mass_sum(k)
        if (initial) then
            msq = sqrts**2
            m = sqrts
            m_max = sqrts
            factor = f1 * f2
            volume = v1 * v2 / (4 * twopi5)
        else
            m_max = sqrts - m_tot + m_min
            call mapping_compute_msq_from_x &
                 (tree%mapping(k), sqrts**2, m_min**2, m_max**2, msq, factor, &
                  x(ix)); ix = ix + 1
            if (msq >= 0) then
                m = sqrt (msq)
                factor = f1 * f2 * factor
                volume = v1 * v2 * sqrts**2 / (4 * twopi2)
                call phs_prt_set_msq (prt(k), msq)
                call phs_prt_set_defined (prt(k))
            else
                ok = .false.
            end if
        end if
        if (ok) then
            msq1 = phs_prt_get_msq (prt(k1));   m1 = sqrt (msq1)
            msq2 = phs_prt_get_msq (prt(k2));   m2 = sqrt (msq2)
            lda = lambda (msq, msq1, msq2)
            if (lda > 0 .and. m > m1 + m2 .and. m <= m_max) then
                rlda = sqrt (lda)
                decay_p(k1) = rlda / (2*m)
                decay_p(k2) = - decay_p(k1)
                factor = rlda / msq * factor
            else
                ok = .false.
            end if
        end if
      end subroutine set_msq_x
  end subroutine phs_tree_set_msq
```

The heart of phase space generation: Now we have the invariant masses, let us generate angles. At each branch, we take a Lorentz transformation and augment it by a boost to the current particle rest frame, and by rotations ϕ and

θ around the z and y axis, respectively. This transformation is passed down to the daughter particles, if present.

⟨*PHS trees: procedures*⟩+≡

```
  subroutine phs_tree_set_angles (tree, prt, factor, decay_p, sqrts, x)
    type(phs_tree_t), intent(inout) :: tree
    type(phs_prt_t), dimension(:), intent(inout) :: prt
    real(default), intent(inout) :: factor
    real(default), dimension(:), intent(in) :: decay_p
    real(default), intent(in) :: sqrts
    real(default), dimension(:), intent(in) :: x
    integer :: ix
    integer(TC) :: k
    ix = 1
    k  = tree%mask_out
    call set_angles_x (tree%branch(k), k)
  contains
    recursive subroutine set_angles_x (b, k, L0)
      type(phs_branch_t), intent(in) :: b
      integer(TC), intent(in) :: k
      type(lorentz_transformation_t), intent(in), optional :: L0
      real(default) :: m, msq, ct, st, phi, f, E, p, bg
      type(lorentz_transformation_t) :: L, LL
      integer(TC) :: k1, k2
      type(vector3_t) :: axis
      p = decay_p(k)
      msq = phs_prt_get_msq (prt(k));  m = sqrt (msq)
      E = sqrt (msq + p**2)
      if (present (L0)) then
         call phs_prt_set_momentum (prt(k), L0 * vector4_moving (E,p,3))
      else
         call phs_prt_set_momentum (prt(k), vector4_moving (E,p,3))
      end if
      call phs_prt_set_defined (prt(k))
      if (b%has_children) then
         k1 = b%daughter(1)
         k2 = b%daughter(2)
         if (m > 0) then
            bg = p / m
         else
            bg = 0
         end if
         phi = x(ix) * twopi;  ix = ix + 1
         call mapping_compute_ct_from_x &
              (tree%mapping(k), sqrts**2, ct, st, f, x(ix));  ix = ix + 1
         factor = factor * f
         if (.not. b%has_friend) then
!            L = boost (bg,3) * rotation (phi,3) * rotation (ct,st,2)
            L = LT_compose_r2_r3_b3 (ct, st, cos(phi), sin(phi), bg)
         else
            LL = boost (-bg,3);  if (present (L0))  LL = LL * inverse(L0)
            axis = space_part ( &
                 LL * phs_prt_get_momentum (prt(tree%branch(k)%friend)) )
            L = boost(bg,3) * rotation_to_2nd (vector3_canonical(3), axis) &
!                 & * rotation(phi,3) * rotation(ct,st,2)
```

```
              * LT_compose_r2_r3_b3 (ct, st, cos(phi), sin(phi), 0._default)
          end if
          if (present (L0))  L = L0 * L
          call set_angles_x (tree%branch(k1), k1, L)
          call set_angles_x (tree%branch(k2), k2, L)
        end if
      end subroutine set_angles_x
    end subroutine phs_tree_set_angles
```

Recover random numbers

For the other channels we want to compute the random numbers that would
have generated the momenta that we already know.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_compute_x_from_momenta
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_compute_x_from_momenta (tree, prt, factor, sqrts, x)
      type(phs_tree_t), intent(inout) :: tree
      type(phs_prt_t), dimension(:), intent(in) :: prt
      real(default), intent(out) :: factor
      real(default), intent(in) :: sqrts
      real(default), dimension(:), intent(inout) :: x
      real(default), dimension(tree%mask_out) :: decay_p
      integer :: n1, n2
      n1 = tree%n_msq
      n2 = n1 + tree%n_angles
      call phs_tree_get_msq &
           (tree, prt, factor, decay_p, sqrts, x(1:n1))
      call phs_tree_get_angles &
           (tree, prt, factor, decay_p, sqrts, x(n1+1:n2))
    end subroutine phs_tree_compute_x_from_momenta
```

The inverse operation follows exactly the same steps. The tree is `inout` be-
cause it contains mappings whose parameters can be reset when the mapping is
applied.

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_get_msq (tree, prt, factor, decay_p, sqrts, x)
      type(phs_tree_t), intent(inout) :: tree
      type(phs_prt_t), dimension(:), intent(in) :: prt
      real(default), intent(out) :: factor
      real(default), dimension(:), intent(out) :: decay_p
      real(default), intent(in) :: sqrts
      real(default), dimension(:), intent(inout) :: x
      integer :: ix
      integer(TC) :: k
      real(default) :: m_tot
      ix = 1
      k  = tree%mask_out
      m_tot = tree%mass_sum(k)
      decay_p(k) = 0.
      if (tree%branch(k)%has_children) then
```

```
          call get_msq_x (tree%branch(k), k, factor, .true.)
       else
          factor = 1
       end if
  contains
    recursive subroutine get_msq_x (b, k, factor, initial)
      type(phs_branch_t), intent(in) :: b
      integer(TC), intent(in) :: k
      real(default), intent(out) :: factor
      logical, intent(in) :: initial
      real(default) :: msq, m, m_min, m_max, msq1, msq2, lda, rlda
      integer(TC) :: k1, k2
      real(default) :: f1, f2
      k1 = b%daughter(1);   k2 = b%daughter(2)
      if (tree%branch(k1)%has_children) then
         call get_msq_x (tree%branch(k1), k1, f1, .false.)
      else
         f1 = 1
      end if
      if (tree%branch(k2)%has_children) then
         call get_msq_x (tree%branch(k2), k2, f2, .false.)
      else
         f2 = 1
      end if
      m_min = tree%mass_sum(k)
      m_max = sqrts - m_tot + m_min
      msq = phs_prt_get_msq (prt(k));   m = sqrt (msq)
      if (initial) then
         factor = f1 * f2
      else
         call mapping_compute_x_from_msq &
              (tree%mapping(k), sqrts**2, m_min**2, m_max**2, msq, factor, &
               x(ix));   ix = ix + 1
         factor = f1 * f2 * factor
      end if
      msq1 = phs_prt_get_msq (prt(k1))
      msq2 = phs_prt_get_msq (prt(k2))
      lda = lambda (msq, msq1, msq2)
      if (lda > 0) then
         rlda = sqrt (lda)
         decay_p(k1) = rlda / (2 * m)
         decay_p(k2) = - decay_p(k1)
         factor = rlda / msq * factor
      else
         decay_p(k1) = 0
         decay_p(k2) = 0
         factor = 0
      end if
    end subroutine get_msq_x
  end subroutine phs_tree_get_msq
```

This subroutine is the most time-critical part of the whole program. There-
fore, we do not exactly parallel the angle generation routine above but make
sure that things get evaluated only if they are really needed, at the expense of

865

readability. Particularly important is to have as few multiplications of Lorentz transformations as possible.

⟨*PHS trees: procedures*⟩+≡

```
subroutine phs_tree_get_angles (tree, prt, factor, decay_p, sqrts, x)
  type(phs_tree_t), intent(inout) :: tree
  type(phs_prt_t), dimension(:), intent(in) :: prt
  real(default), intent(inout) :: factor
  real(default), dimension(:), intent(in) :: decay_p
  real(default), intent(in) :: sqrts
  real(default), dimension(:), intent(out) :: x
  integer :: ix
  integer(TC) :: k
  ix = 1
  k  = tree%mask_out
  if (tree%branch(k)%has_children)  call get_angles_x (tree%branch(k), k)
contains
  recursive subroutine get_angles_x (b, k, ct0, st0, phi0, L0)
    type(phs_branch_t), intent(in) :: b
    integer(TC), intent(in) :: k
    real(default), intent(in), optional :: ct0, st0, phi0
    type(lorentz_transformation_t), intent(in), optional :: L0
    real(default) :: cp0, sp0, m, msq, ct, st, phi, bg, f
    type(lorentz_transformation_t) :: L, LL
    type(vector4_t) :: p1, pf
    type(vector3_t) :: n, axis
    integer(TC) :: k1, k2, kf
    logical :: has_friend, need_L
    k1 = b%daughter(1)
    k2 = b%daughter(2)
    kf = b%friend
    has_friend = b%has_friend
    if (present(L0)) then
       p1 = L0 * phs_prt_get_momentum (prt(k1))
       if (has_friend)  pf = L0 * phs_prt_get_momentum (prt(kf))
    else
       p1 = phs_prt_get_momentum (prt(k1))
       if (has_friend)  pf = phs_prt_get_momentum (prt(kf))
    end if
    if (present(phi0)) then
       cp0 = cos (phi0)
       sp0 = sin (phi0)
    end if
    msq = phs_prt_get_msq (prt(k));  m = sqrt (msq)
    if (m > 0) then
       bg = decay_p(k) / m
    else
       bg = 0
    end if
    if (has_friend) then
       if (present (phi0)) then
          axis = axis_from_p_r3_r2_b3 (pf, cp0, -sp0, ct0, -st0, -bg)
          LL = rotation_to_2nd (axis, vector3_canonical (3)) &
                  * LT_compose_r3_r2_b3 (cp0, -sp0, ct0, -st0, -bg)
       else
```

866

```
            axis = axis_from_p_b3 (pf, -bg)
            LL = rotation_to_2nd (axis, vector3_canonical(3))
            if (bg /= 0)  LL = LL * boost(-bg, 3)
         end if
         n = space_part (LL * p1)
      else if (present (phi0)) then
         n = axis_from_p_r3_r2_b3 (p1, cp0, -sp0, ct0, -st0, -bg)
      else
         n = axis_from_p_b3 (p1, -bg)
      end if
      phi = azimuthal_angle (n)
      x(ix) = phi / twopi;  ix = ix + 1
      ct = polar_angle_ct (n)
      st = sqrt (1 - ct**2)
      call mapping_compute_x_from_ct (tree%mapping(k), sqrts**2, ct, f, &
           x(ix));  ix = ix + 1
      factor = factor * f
      if (tree%branch(k1)%has_children .or. tree%branch(k2)%has_children) then
         need_L = .true.
         if (has_friend) then
            if (present (L0)) then
               L = LL * L0
            else
               L = LL
            end if
         else if (present (L0)) then
            L = LT_compose_r3_r2_b3 (cp0, -sp0, ct0, -st0, -bg) * L0
         else if (present (phi0)) then
            L = LT_compose_r3_r2_b3 (cp0, -sp0, ct0, -st0, -bg)
         else if (bg /= 0) then
            L = boost(-bg, 3)
         else
            need_L = .false.
         end if
         if (need_L) then
            if (tree%branch(k1)%has_children) &
                 call get_angles_x (tree%branch(k1), k1, ct, st, phi, L)
            if (tree%branch(k2)%has_children) &
                 call get_angles_x (tree%branch(k2), k2, ct, st, phi, L)
         else
            if (tree%branch(k1)%has_children) &
                 call get_angles_x (tree%branch(k1), k1, ct, st, phi)
            if (tree%branch(k2)%has_children) &
                 call get_angles_x (tree%branch(k2), k2, ct, st, phi)
         end if
      end if
    end subroutine get_angles_x
  end subroutine phs_tree_get_angles
```

Auxiliary stuff

This calculates all momenta that are not yet known by summing up daughter
particle momenta. The external particles must be known. Only composite

particles not yet known are calculated.

⟨*PHS trees: public*⟩+≡

```
public :: phs_tree_combine_particles
```

⟨*PHS trees: procedures*⟩+≡

```
subroutine phs_tree_combine_particles (tree, prt)
  type(phs_tree_t), intent(in) :: tree
  type(phs_prt_t), dimension(:), intent(inout) :: prt
  call combine_particles_x (tree%mask_out)
contains
  recursive subroutine combine_particles_x (k)
    integer(TC), intent(in) :: k
    integer :: k1, k2
    if (tree%branch(k)%has_children) then
       k1 = tree%branch(k)%daughter(1);  k2 = tree%branch(k)%daughter(2)
       call combine_particles_x (k1)
       call combine_particles_x (k2)
       if (.not. phs_prt_is_defined (prt(k))) then
          call phs_prt_combine (prt(k), prt(k1), prt(k2))
       end if
    end if
  end subroutine combine_particles_x
end subroutine phs_tree_combine_particles
```

12.3   The phase-space forest

Simply stated, a phase-space forest is a collection of phase-space trees. More precisely, a `phs_forest` object contains all parameterizations of phase space that WHIZARD will use for a single hard process, prepared in the form of `phs_tree` objects. This is suitable for evaluation by the VAMP integration package: each parameterization (tree) is a valid channel in the multi-channel adaptive integration, and each variable in a tree corresponds to an integration dimension, defined by an appropriate mapping of the $(0, 1)$ interval to the allowed range of the integration variable.

The trees are grouped in groves. The trees (integration channels) within a grove share a common weight, assuming that they are related by some approximate symmetry.

Trees/channels that are related by an exact symmetry are connected by an array of equivalences; each equivalence object holds the data that relate one channel to another.

The phase-space setup, i.e., the detailed structure of trees and forest, are read from a file. Therefore, this module also contains the syntax definition and the parser needed for interpreting this file.

⟨phs_forests.f90⟩≡
  ⟨*File header*⟩

  module phs_forests

  ⟨*Use kinds*⟩
    use kinds, only: TC !NODEP!
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use vamp_equivalences !NODEP!
    use permutations
    use ifiles
    use syntax_rules
    use lexers
    use parser
    use models
    use flavors
    use interactions
    use mappings
    use phs_trees

  ⟨*Standard module head*⟩

  ⟨*PHS forests: public*⟩

  ⟨*PHS forests: types*⟩

  ⟨*PHS forests: interfaces*⟩

  ⟨*PHS forests: variables*⟩

```
    contains
```

⟨*PHS forests: procedures*⟩

```
    end module phs_forests
```

12.3.1   Phase-space setup parameters

This transparent container holds the parameters that the algorithm needs for
phase-space setup, with reasonable defaults.

The threshold mass (for considering a particle as effectively massless) is
specified separately for s- and t-channel. The default is to treat W and Z bosons
as massive in the s-channel, but as massless in the t-channel. The b-quark is
treated always massless, the t-quark always massive.

⟨*PHS forests: public*⟩≡
```
    public :: phs_parameters_t
```

⟨*PHS forests: types*⟩≡
```
    type :: phs_parameters_t
       real(default) :: sqrts = 0
       real(default) :: m_threshold_s = 50._default
       real(default) :: m_threshold_t = 100._default
       integer :: off_shell = 1
       integer :: t_channel = 2
    end type phs_parameters_t
```

Write phase-space parameters to file.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_parameters_write
```

⟨*PHS forests: procedures*⟩≡
```
    subroutine phs_parameters_write (phs_par, unit)
      type(phs_parameters_t), intent(in) :: phs_par
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, *) "  sqrts         = ", phs_par%sqrts
      write (u, *) "  m_threshold_s = ", phs_par%m_threshold_s
      write (u, *) "  m_threshold_t = ", phs_par%m_threshold_t
      write (u, *) "  off_shell = ", phs_par%off_shell
      write (u, *) "  t_channel = ", phs_par%t_channel
    end subroutine phs_parameters_write
```

Read phase-space parameters from file.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_parameters_read
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_parameters_read (phs_par, unit)
      type(phs_parameters_t), intent(out) :: phs_par
      integer, intent(in) :: unit
      character(20) :: dummy
      character :: equals
```

```
      read (unit, *)  dummy, equals, phs_par%sqrts
      read (unit, *)  dummy, equals, phs_par%m_threshold_s
      read (unit, *)  dummy, equals, phs_par%m_threshold_t
      read (unit, *)  dummy, equals, phs_par%off_shell
      read (unit, *)  dummy, equals, phs_par%t_channel
    end subroutine phs_parameters_read
```

Comparison.

⟨*PHS forests: interfaces*⟩≡
```
    interface operator(==)
       module procedure phs_parameters_eq
    end interface
    interface operator(/=)
       module procedure phs_parameters_ne
    end interface
```

⟨*PHS forests: procedures*⟩+≡
```
    function phs_parameters_eq (phs_par1, phs_par2) result (equal)
      logical :: equal
      type(phs_parameters_t), intent(in) :: phs_par1, phs_par2
      equal = phs_par1%sqrts == phs_par2%sqrts &
           .and. phs_par1%m_threshold_s == phs_par2%m_threshold_s &
           .and. phs_par1%m_threshold_t == phs_par2%m_threshold_t &
           .and. phs_par1%off_shell == phs_par2%off_shell &
           .and. phs_par1%t_channel == phs_par2%t_channel
    end function phs_parameters_eq

    function phs_parameters_ne (phs_par1, phs_par2) result (ne)
      logical :: ne
      type(phs_parameters_t), intent(in) :: phs_par1, phs_par2
      ne = phs_par1%sqrts /= phs_par2%sqrts &
           .or. phs_par1%m_threshold_s /= phs_par2%m_threshold_s &
           .or. phs_par1%m_threshold_t /= phs_par2%m_threshold_t &
           .or. phs_par1%off_shell /= phs_par2%off_shell &
           .or. phs_par1%t_channel /= phs_par2%t_channel
    end function phs_parameters_ne
```

12.3.2   Equivalences

This type holds information about equivalences between phase-space trees. We make a linked list, where each node contains the two trees which are equivalent and the corresponding permutation of external particles. Two more arrays are to be filled: The permutation of mass variables and the permutation of angular variables, where the signature indicates a necessary exchange of daughter branches.

⟨*PHS forests: types*⟩+≡
```
    type :: equivalence_t
       private
       integer :: left, right
       type(permutation_t) :: perm
       type(permutation_t) :: msq_perm, angle_perm
       logical, dimension(:), allocatable :: angle_sig
```

```
      type(equivalence_t), pointer :: next => null ()
   end type equivalence_t
```

⟨*PHS forests: types*⟩+≡
```
   type :: equivalence_list_t
      private
      integer :: length = 0
      type(equivalence_t), pointer :: first => null ()
      type(equivalence_t), pointer :: last => null ()
   end type equivalence_list_t
```

Append an equivalence to the list

⟨*PHS forests: procedures*⟩+≡
```
   subroutine equivalence_list_add (eql, left, right, perm)
     type(equivalence_list_t), intent(inout) :: eql
     integer, intent(in) :: left, right
     type(permutation_t), intent(in) :: perm
     type(equivalence_t), pointer :: eq
     allocate (eq)
     eq%left = left
     eq%right = right
     eq%perm = perm
     if (associated (eql%last)) then
        eql%last%next => eq
     else
        eql%first => eq
     end if
     eql%last => eq
     eql%length = eql%length + 1
   end subroutine equivalence_list_add
```

Delete the list contents. Has to be pure because it is called from an elemental subroutine.

⟨*PHS forests: procedures*⟩+≡
```
   pure subroutine equivalence_list_final (eql)
     type(equivalence_list_t), intent(inout) :: eql
     type(equivalence_t), pointer :: eq
     do while (associated (eql%first))
        eq => eql%first
        eql%first => eql%first%next
        deallocate (eq)
     end do
     eql%last => null ()
     eql%length = 0
   end subroutine equivalence_list_final
```

Make a deep copy of the equivalence list. This allows for deep copies of groves and forests.

⟨*PHS forests: interfaces*⟩+≡
```
   interface assignment(=)
      module procedure equivalence_list_assign
   end interface
```

⟨*PHS forests: procedures*⟩+≡

```
subroutine equivalence_list_assign (eql_out, eql_in)
  type(equivalence_list_t), intent(out) :: eql_out
  type(equivalence_list_t), intent(in) :: eql_in
  type(equivalence_t), pointer :: eq, eq_copy
  eq => eql_in%first
  do while (associated (eq))
     allocate (eq_copy)
     eq_copy = eq
     eq_copy%next => null ()
     if (associated (eql_out%first)) then
        eql_out%last%next => eq_copy
     else
        eql_out%first => eq_copy
     end if
     eql_out%last => eq_copy
     eq => eq%next
  end do
end subroutine equivalence_list_assign
```

The number of list entries

⟨*PHS forests: procedures*⟩+≡

```
elemental function equivalence_list_length (eql) result (length)
  integer :: length
  type(equivalence_list_t), intent(in) :: eql
  length = eql%length
end function equivalence_list_length
```

Recursively write the equivalences list

⟨*PHS forests: procedures*⟩+≡

```
subroutine equivalence_list_write (eql, unit)
  type(equivalence_list_t), intent(in) :: eql
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  if (associated (eql%first)) then
     call equivalence_write_rec (eql%first, u)
  else
     write (u, *) " [empty]"
  end if
contains
  recursive subroutine equivalence_write_rec (eq, u)
    type(equivalence_t), intent(in) :: eq
    integer, intent(in) :: u
    integer :: i
    write (u, "(1x,A,1x,I5,1x,I5,5x,A)", advance="no") &
         "Equivalence:", eq%left, eq%right, "Final state permutation:"
    call permutation_write (eq%perm, u)
    write (u, "(1x,12x,1x,A,1x)", advance="no") &
         "         msq permutation:   "
    call permutation_write (eq%msq_perm, u)
```

```
      write (u, "(1x,12x,1x,A,1x)", advance="no") &
           "         angle permutation:"
      call permutation_write (eq%angle_perm, u)
      write (u, "(1x,12x,1x,26x)", advance="no")
      do i = 1, size (eq%angle_sig)
         if (eq%angle_sig(i)) then
            write (u, "(1x,A)", advance="no") "+"
         else
            write (u, "(1x,A)", advance="no") "-"
         end if
      end do
      write (u, *)
      if (associated (eq%next))  call equivalence_write_rec (eq%next, u)
    end subroutine equivalence_write_rec
  end subroutine equivalence_list_write
```

12.3.3 Groves

A grove is a group of trees (phase-space channels) that share a common weight
in the integration. Within a grove, channels can be declared equivalent, so they
also share their integration grids (up to symmetries). The grove contains a list
of equivalences. The `tree_count_offset` is the total number of trees of the
preceding groves; when the trees are counted per forest (integration channels),
the offset has to be added to all tree indices.

⟨*PHS forests: types*⟩+≡
```
    type :: phs_grove_t
       private
       integer :: tree_count_offset
       type(phs_tree_t), dimension(:), allocatable :: tree
       type(equivalence_list_t) :: equivalence_list
    end type phs_grove_t
```

Call `phs_tree_init` which is also elemental:

⟨*PHS forests: procedures*⟩+≡
```
    elemental subroutine phs_grove_init &
         (grove, n_trees, n_in, n_out, n_masses, n_angles)
      type(phs_grove_t), intent(inout) :: grove
      integer, intent(in) :: n_trees, n_in, n_out, n_masses, n_angles
      grove%tree_count_offset = 0
      allocate (grove%tree (n_trees))
      call phs_tree_init (grove%tree, n_in, n_out, n_masses, n_angles)
    end subroutine phs_grove_init
```

The trees do not have pointer components, thus no call to `phs_tree_final`:

⟨*PHS forests: procedures*⟩+≡
```
    elemental subroutine phs_grove_final (grove)
      type(phs_grove_t), intent(inout) :: grove
      deallocate (grove%tree)
      call equivalence_list_final (grove%equivalence_list)
    end subroutine phs_grove_final
```

Deep copy.

⟨PHS forests: interfaces⟩+≡
```
interface assignment(=)
   module procedure phs_grove_assign0
   module procedure phs_grove_assign1
end interface
```

⟨PHS forests: procedures⟩+≡
```
subroutine phs_grove_assign0 (grove_out, grove_in)
  type(phs_grove_t), intent(out) :: grove_out
  type(phs_grove_t), intent(in) :: grove_in
  grove_out%tree_count_offset = grove_in%tree_count_offset
  if (allocated (grove_in%tree)) then
     allocate (grove_out%tree (size (grove_in%tree)))
     grove_out%tree = grove_in%tree
  end if
  grove_out%equivalence_list = grove_in%equivalence_list
end subroutine phs_grove_assign0

subroutine phs_grove_assign1 (grove_out, grove_in)
  type(phs_grove_t), dimension(:), intent(out) :: grove_out
  type(phs_grove_t), dimension(:), intent(in) :: grove_in
  integer :: i
  do i = 1, size (grove_in)
     call phs_grove_assign0 (grove_out(i), grove_in(i))
  end do
end subroutine phs_grove_assign1
```

12.3.4   The forest type

This is a collection of trees and associated particles. In a given tree, each branch code corresponds to a particle in the `prt` array. Furthermore, we have an array of mass sums which is independent of the decay tree and of the particular event. The mappings directly correspond to the decay trees, and the decay groves collect the trees in classes. The permutation list consists of all permutations of outgoing particles that map the decay forest onto itself.

The particle codes `flv` (one for each external particle) are needed for determining masses and such. The trees and associated information are collected in the `grove` array, together with a lookup table that associates tree indices to groves. Finally, the `prt` array serves as workspace for phase-space evaluation.

⟨PHS forests: public⟩+≡
```
public :: phs_forest_t
```

⟨PHS forests: types⟩+≡
```
type :: phs_forest_t
   private
   integer :: n_in, n_out, n_tot
   integer :: n_masses, n_angles, n_dimensions
   integer :: n_trees, n_equivalences
   type(flavor_t), dimension(:), allocatable :: flv
   type(phs_grove_t), dimension(:), allocatable :: grove
```

```
          integer, dimension(:), allocatable :: grove_lookup
          type(phs_prt_t), dimension(:), allocatable :: prt_in
          type(phs_prt_t), dimension(:), allocatable :: prt_out
          type(phs_prt_t), dimension(:), allocatable :: prt
       end type phs_forest_t
```

The initialization merely allocates memory. We have to know how many trees
there are in each grove, so we can initialize everything. The number of groves
is the size of the `n_tree` array.

In the `grove_lookup` table we store the grove index that belongs to each
absolute tree index. The difference between the absolute index and the relative
(to the grove) index is stored, for each grove, as `tree_count_offset`.

The particle array is allocated according to the total number of branches
each tree has, but not filled.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_init
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_forest_init (forest, n_tree, n_in, n_out)
      type(phs_forest_t), intent(inout) :: forest
      integer, dimension(:), intent(in) :: n_tree
      integer, intent(in) :: n_in, n_out
      integer :: g, count
      forest%n_in = n_in
      forest%n_out = n_out
      forest%n_tot = n_in + n_out
      forest%n_masses = max (n_out - 2, 0)
      forest%n_angles = max (2*n_out - 2, 0)
      forest%n_dimensions = forest%n_masses + forest%n_angles
      forest%n_trees = sum (n_tree)
      forest%n_equivalences = 0
      allocate (forest%grove (size (n_tree)))
      call phs_grove_init &
           (forest%grove, n_tree, n_in, n_out, forest%n_masses, forest%n_angles)
      allocate (forest%grove_lookup (forest%n_trees))
      count = 0
      do g = 1, size (forest%grove)
         forest%grove(g)%tree_count_offset = count
         forest%grove_lookup (count+1:count+n_tree(g)) = g
         count = count + n_tree(g)
      end do
      allocate (forest%prt_in  (n_in))
      allocate (forest%prt_out (n_out))
      allocate (forest%prt (2**forest%n_tot - 1))
    end subroutine phs_forest_init
```

The grove finalizer is called because it contains the equivalence list:

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_final
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_forest_final (forest)
      type(phs_forest_t), intent(inout) :: forest
      if (allocated (forest%grove)) then
```

```
        call phs_grove_final (forest%grove)
        deallocate (forest%grove)
      end if
    if (allocated (forest%grove_lookup))  deallocate (forest%grove_lookup)
    if (allocated (forest%prt))  deallocate (forest%prt)
  end subroutine phs_forest_final
```

12.3.5 Screen output

Write the particles that are non-null, then the trees which point to them:

⟨PHS forests: public⟩+≡
```
    public :: phs_forest_write
```

⟨PHS forests: procedures⟩+≡
```
    subroutine phs_forest_write (forest, unit)
      type(phs_forest_t), intent(in) :: forest
      integer, intent(in), optional :: unit
      integer :: u
      integer :: i, g
      u = output_unit (unit);  if (u < 0)  return
      write (u, *) "Phase space forest:"
      write (u, *) "n_in  = ", forest%n_in
      write (u, *) "n_out = ", forest%n_out
      write (u, *) "n_tot = ", forest%n_tot
      write (u, *) "n_masses = ", forest%n_masses
      write (u, *) "n_angles = ", forest%n_angles
      write (u, *) "n_dim    = ", forest%n_dimensions
      write (u, *) "n_trees  = ", forest%n_trees
      write (u, *) "n_equiv  = ", forest%n_equivalences
      write (u, "(1x,A)", advance="no") "flavors  ="
      if (allocated (forest%flv)) then
         do i = 1, size (forest%flv)
            write (u, "(1x,I6)", advance="no")  flavor_get_pdg (forest%flv(i))
         end do
         write (u, *)
      else
         write (u, *) "[empty]"
      end if
      write (u, *) "Groves and trees:"
      if (allocated (forest%grove)) then
         do g = 1, size (forest%grove)
            write (u, "(1x,A,1x,I4)") "Grove ", g
            call phs_grove_write (forest%grove(g), unit)
         end do
      else
         write (u, *) "  [empty]"
      end if
      write (u, *) "Total number of equivalences: ", forest%n_equivalences
      write (u, *)
      write (u, *) "Incoming particles:"
      if (allocated (forest%prt_in)) then
         if (any (phs_prt_is_defined (forest%prt_in))) then
            do i = 1, size (forest%prt_in)
```

877

```fortran
                  if (phs_prt_is_defined (forest%prt_in(i))) then
                     write (u, *)  "Particle", i
                     call phs_prt_write (forest%prt_in(i), u)
                  end if
               end do
            else
               write (u, "(3x,A)")  "[all undefined]"
            end if
         else
            write (u, *)  "  [empty]"
         end if
         write (u, *)
         write (u, *) "Outgoing particles:"
         if (allocated (forest%prt_out)) then
            if (any (phs_prt_is_defined (forest%prt_out))) then
               do i = 1, size (forest%prt_out)
                  if (phs_prt_is_defined (forest%prt_out(i))) then
                     write (u, *)  "Particle", i
                     call phs_prt_write (forest%prt_out(i), u)
                  end if
               end do
            else
               write (u, "(3x,A)")  "[all undefined]"
            end if
         else
            write (u, *)  "  [empty]"
         end if
         write (u, *)
         write (u, *) "Tree particles:"
         if (allocated (forest%prt)) then
            if (any (phs_prt_is_defined (forest%prt))) then
               do i = 1, size (forest%prt)
                  if (phs_prt_is_defined (forest%prt(i))) then
                     write (u, *)  "Particle", i
                     call phs_prt_write (forest%prt(i), u)
                  end if
               end do
            else
               write (u, "(3x,A)")  "[all undefined]"
            end if
         else
            write (u, *)  "  [empty]"
         end if
       end subroutine phs_forest_write

       subroutine phs_grove_write (grove, unit)
         type(phs_grove_t), intent(in) :: grove
         integer, intent(in), optional :: unit
         integer :: u
         integer :: t
         u = output_unit (unit);  if (u < 0)  return
         do t = 1, size (grove%tree)
            write (u, "(1x,A,1x,I4)") "Tree  ", t
            call phs_tree_write (grove%tree(t), unit)
```

```
      end do
      write (u, "(1x,A)") "Equivalence list:"
      call equivalence_list_write (grove%equivalence_list, unit)
    end subroutine phs_grove_write
```

Deep copy.

⟨PHS forests: public⟩+≡
```
    public :: assignment(=)
```

⟨PHS forests: interfaces⟩+≡
```
    interface assignment(=)
       module procedure phs_forest_assign
    end interface
```

⟨PHS forests: procedures⟩+≡
```
    subroutine phs_forest_assign (forest_out, forest_in)
      type(phs_forest_t), intent(out) :: forest_out
      type(phs_forest_t), intent(in) :: forest_in
      forest_out%n_in  = forest_in%n_in
      forest_out%n_out = forest_in%n_out
      forest_out%n_tot = forest_in%n_tot
      forest_out%n_masses = forest_in%n_masses
      forest_out%n_angles = forest_in%n_angles
      forest_out%n_dimensions  = forest_in%n_dimensions
      forest_out%n_trees  = forest_in%n_trees
      forest_out%n_equivalences  = forest_in%n_equivalences
      if (allocated (forest_in%flv)) then
         allocate (forest_out%flv (size (forest_in%flv)))
         forest_out%flv = forest_in%flv
      end if
      if (allocated (forest_in%grove)) then
         allocate (forest_out%grove (size (forest_in%grove)))
         forest_out%grove = forest_in%grove
      end if
      if (allocated (forest_in%grove_lookup)) then
         allocate (forest_out%grove_lookup (size (forest_in%grove_lookup)))
         forest_out%grove_lookup = forest_in%grove_lookup
      end if
      if (allocated (forest_in%prt_in)) then
         allocate (forest_out%prt_in (size (forest_in%prt_in)))
         forest_out%prt_in = forest_in%prt_in
      end if
      if (allocated (forest_in%prt_out)) then
         allocate (forest_out%prt_out (size (forest_in%prt_out)))
         forest_out%prt_out = forest_in%prt_out
      end if
      if (allocated (forest_in%prt)) then
         allocate (forest_out%prt (size (forest_in%prt)))
         forest_out%prt = forest_in%prt
      end if
    end subroutine phs_forest_assign
```

12.3.6 Accessing contents

Get the number of integration parameters

⟨*PHS forests: public*⟩+≡
```
public :: phs_forest_get_n_parameters
```

⟨*PHS forests: procedures*⟩+≡
```
function phs_forest_get_n_parameters (forest) result (n)
  integer :: n
  type(phs_forest_t), intent(in) :: forest
  n = forest%n_dimensions
end function phs_forest_get_n_parameters
```

Get the number of integration channels

⟨*PHS forests: public*⟩+≡
```
public :: phs_forest_get_n_channels
```

⟨*PHS forests: procedures*⟩+≡
```
function phs_forest_get_n_channels (forest) result (n)
  integer :: n
  type(phs_forest_t), intent(in) :: forest
  n = forest%n_trees
end function phs_forest_get_n_channels
```

Get the number of groves

⟨*PHS forests: public*⟩+≡
```
public :: phs_forest_get_n_groves
```

⟨*PHS forests: procedures*⟩+≡
```
function phs_forest_get_n_groves (forest) result (n)
  integer :: n
  type(phs_forest_t), intent(in) :: forest
  n = size (forest%grove)
end function phs_forest_get_n_groves
```

Get the index bounds for a specific grove.

⟨*PHS forests: public*⟩+≡
```
public :: phs_forest_get_grove_bounds
```

⟨*PHS forests: procedures*⟩+≡
```
subroutine phs_forest_get_grove_bounds (forest, g, i0, i1, n)
  type(phs_forest_t), intent(in) :: forest
  integer, intent(in) :: g
  integer, intent(out) :: i0, i1, n
  n = size (forest%grove(g)%tree)
  i0 = forest%grove(g)%tree_count_offset + 1
  i1 = forest%grove(g)%tree_count_offset + n
end subroutine phs_forest_get_grove_bounds
```

Get the number of equivalences

⟨*PHS forests: public*⟩+≡
```
public :: phs_forest_get_n_equivalences
```

⟨PHS forests: procedures⟩+≡

```
function phs_forest_get_n_equivalences (forest) result (n)
  integer :: n
  type(phs_forest_t), intent(in) :: forest
  n = forest%n_equivalences
end function phs_forest_get_n_equivalences
```

12.3.7 Read the phase space setup from file

The phase space setup is stored in a file. The file may be generated by the `cascades` module below, or by other means. This file has to be read and parsed to create the PHS forest as the internal phase-space representation.

Create lexer and syntax:

⟨PHS forests: procedures⟩+≡

```
subroutine define_phs_forest_syntax (ifile)
  type(ifile_t) :: ifile
  call ifile_append (ifile, "SEQ phase_space_list = process_phase_space*")
  call ifile_append (ifile, "SEQ process_phase_space = " &
       // "process_def process_header phase_space")
  call ifile_append (ifile, "SEQ process_def = process process_list")
  call ifile_append (ifile, "KEY process")
  call ifile_append (ifile, "LIS process_list = process_tag*")
  call ifile_append (ifile, "IDE process_tag")
  call ifile_append (ifile, "SEQ process_header = " &
       // "md5sum_process = md5sum " &
       // "md5sum_model = md5sum " &
       // "md5sum_parameters = md5sum " &
       // "sqrts = real " &
       // "m_threshold_s = real " &
       // "m_threshold_t = real " &
       // "off_shell = integer " &
       // "t_channel = integer ")
  call ifile_append (ifile, "KEY '='")
  call ifile_append (ifile, "KEY md5sum_process")
  call ifile_append (ifile, "KEY md5sum_model")
  call ifile_append (ifile, "KEY md5sum_parameters")
  call ifile_append (ifile, "KEY sqrts")
  call ifile_append (ifile, "KEY m_threshold_s")
  call ifile_append (ifile, "KEY m_threshold_t")
  call ifile_append (ifile, "KEY off_shell")
  call ifile_append (ifile, "KEY t_channel")
  call ifile_append (ifile, "QUO md5sum = '""' ... '""'")
  call ifile_append (ifile, "REA real")
  call ifile_append (ifile, "INT integer")
  call ifile_append (ifile, "SEQ phase_space = grove_def+")
  call ifile_append (ifile, "SEQ grove_def = grove tree_def+")
  call ifile_append (ifile, "KEY grove")
  call ifile_append (ifile, "SEQ tree_def = tree bincodes mapping*")
  call ifile_append (ifile, "KEY tree")
  call ifile_append (ifile, "SEQ bincodes = bincode+")
  call ifile_append (ifile, "INT bincode")
  call ifile_append (ifile, "SEQ mapping = map bincode channel pdg")
```

```
          call ifile_append (ifile, "KEY map")
          call ifile_append (ifile, "ALT channel = s_channel | t_channel | u_channel | collinear | infra
          call ifile_append (ifile, "KEY s_channel")
          ! call ifile_append (ifile, "KEY t_channel")
          call ifile_append (ifile, "KEY u_channel")
          call ifile_append (ifile, "KEY collinear")
          call ifile_append (ifile, "KEY infrared")
          call ifile_append (ifile, "KEY radiation")
          call ifile_append (ifile, "INT pdg")
    end subroutine define_phs_forest_syntax
```

The model-file syntax and lexer are fixed, therefore stored as module variables:

⟨*PHS forests: variables*⟩≡
```
    type(syntax_t), target, save :: syntax_phs_forest
```

⟨*PHS forests: public*⟩+≡
```
    public :: syntax_phs_forest_init
```
⟨*PHS forests: procedures*⟩+≡
```
    subroutine syntax_phs_forest_init ()
      type(ifile_t) :: ifile
      call define_phs_forest_syntax (ifile)
      call syntax_init (syntax_phs_forest, ifile)
      call ifile_final (ifile)
    end subroutine syntax_phs_forest_init
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine lexer_init_phs_forest (lexer)
      type(lexer_t), intent(out) :: lexer
      call lexer_init (lexer, &
           comment_chars = "#!", &
           quote_chars = '"', &
           quote_match = '"', &
           single_chars = "", &
           special_class = (/ "=" /) , &
           keyword_list = syntax_get_keyword_list_ptr (syntax_phs_forest))
    end subroutine lexer_init_phs_forest
```

⟨*PHS forests: public*⟩+≡
```
    public :: syntax_phs_forest_final
```
⟨*PHS forests: procedures*⟩+≡
```
    subroutine syntax_phs_forest_final ()
      call syntax_final (syntax_phs_forest)
    end subroutine syntax_phs_forest_final
```

⟨*PHS forests: public*⟩+≡
```
    public :: syntax_phs_forest_write
```
⟨*PHS forests: procedures*⟩+≡
```
    subroutine syntax_phs_forest_write (unit)
      integer, intent(in), optional :: unit
      call syntax_write (syntax_phs_forest, unit)
    end subroutine syntax_phs_forest_write
```

The concrete parser and interpreter. Generate an input stream for the external
`unit`, read the parse tree (with given `syntax` and `lexer`) from this stream, and
transfer the contents of the parse tree to the PHS `forest`.

We look for the matching `process` tag, count groves and trees for initializing
the `forest`, and fill the trees.

If the optional parameters are set, compare the parameters stored in the file
to those. Set `match` true if everything agrees.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_read
```

⟨*PHS forests: interfaces*⟩+≡
```
    interface phs_forest_read
       module procedure phs_forest_read_file
       module procedure phs_forest_read_unit
       module procedure phs_forest_read_parse_tree
    end interface
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_forest_read_file &
         (forest, filename, process_id, n_in, n_out, model, found, &
          md5sum_process, md5sum_model, md5sum_parameters, phs_par, match)
      type(phs_forest_t), intent(out) :: forest
      type(string_t), intent(in) :: filename
      type(string_t), intent(in) :: process_id
      integer, intent(in) :: n_in, n_out
      type(model_t), intent(in), target :: model
      logical, intent(out) :: found
      character(32), intent(in), optional :: &
           md5sum_process, md5sum_model, md5sum_parameters
      type(phs_parameters_t), intent(in), optional :: phs_par
      logical, intent(out), optional :: match
      type(parse_tree_t), target :: parse_tree
      type(stream_t), target :: stream
      type(lexer_t) :: lexer
      call lexer_init_phs_forest (lexer)
      call stream_init (stream, char (filename))
      call lexer_assign_stream (lexer, stream)
      call parse_tree_init (parse_tree, syntax_phs_forest, lexer)
      call phs_forest_read (forest, parse_tree, &
           process_id, n_in, n_out, model, found, &
           md5sum_process, md5sum_model, md5sum_parameters, phs_par, match)
      call stream_final (stream)
      call lexer_final (lexer)
      call parse_tree_final (parse_tree)
    end subroutine phs_forest_read_file

    subroutine phs_forest_read_unit &
         (forest, unit, process_id, n_in, n_out, model, found, &
          md5sum_process, md5sum_model, md5sum_parameters, phs_par, match)
      type(phs_forest_t), intent(out) :: forest
      integer, intent(in) :: unit
      type(string_t), intent(in) :: process_id
      integer, intent(in) :: n_in, n_out
      type(model_t), intent(in), target :: model
```

```
      logical, intent(out) :: found
      character(32), intent(in), optional :: &
           md5sum_process, md5sum_model, md5sum_parameters
      type(phs_parameters_t), intent(in), optional :: phs_par
      logical, intent(out), optional :: match
      type(parse_tree_t), target :: parse_tree
      type(stream_t), target :: stream
      type(lexer_t) :: lexer
      call lexer_init_phs_forest (lexer)
      call stream_init (stream, unit)
      call lexer_assign_stream (lexer, stream)
      call parse_tree_init (parse_tree, syntax_phs_forest, lexer)
      call phs_forest_read (forest, parse_tree, &
           process_id, n_in, n_out, model, found, &
           md5sum_process, md5sum_model, md5sum_parameters, phs_par, match)
      call stream_final (stream)
      call lexer_final (lexer)
      call parse_tree_final (parse_tree)
    end subroutine phs_forest_read_unit

    subroutine phs_forest_read_parse_tree &
         (forest, parse_tree, process_id, n_in, n_out, model, found, &
          md5sum_process, md5sum_model, md5sum_parameters, phs_par, match)
      type(phs_forest_t), intent(out) :: forest
      type(parse_tree_t), intent(in), target :: parse_tree
      type(string_t), intent(in) :: process_id
      integer, intent(in) :: n_in, n_out
      type(model_t), intent(in), target :: model
      logical, intent(out) :: found
      character(32), intent(in), optional :: &
           md5sum_process, md5sum_model, md5sum_parameters
      type(phs_parameters_t), intent(in), optional :: phs_par
      logical, intent(out), optional :: match
      type(parse_node_t), pointer :: node_header, node_phs, node_grove
      integer :: n_grove, g
      integer, dimension(:), allocatable :: n_tree
      integer :: t
!     call parse_tree_write (parse_tree)
      node_header => parse_tree_get_process_ptr (parse_tree, process_id)
      found = associated (node_header);  if (.not. found)  return
      if (present (match)) then
         call phs_forest_check_input (node_header, &
              md5sum_process, md5sum_model, md5sum_parameters, phs_par, match)
         if (.not. match)  return
      end if
      node_phs => parse_node_get_next_ptr (node_header)
      n_grove = parse_node_get_n_sub (node_phs)
      allocate (n_tree (n_grove))
      do g = 1, n_grove
         node_grove => parse_node_get_sub_ptr (node_phs, g)
         n_tree(g) = parse_node_get_n_sub (node_grove) - 1
      end do
      call phs_forest_init (forest, n_tree, n_in, n_out)
      do g = 1, n_grove
```

```
            node_grove => parse_node_get_sub_ptr (node_phs, g)
            do t = 1, n_tree(g)
               call phs_tree_set (forest%grove(g)%tree(t), &
                     parse_node_get_sub_ptr (node_grove, t+1), model)
            end do
         end do
    end subroutine phs_forest_read_parse_tree
```

Check the input for consistency. If any MD5 sum or phase-space parameter disagrees, the phase-space file cannot be used. The MD5 sum checks are skipped if the stored MD5 sum is empty.

⟨*PHS forests: procedures*⟩+≡

```
    subroutine phs_forest_check_input (pn_header, &
         md5sum_process, md5sum_model, md5sum_parameters, phs_par, match)
      type(parse_node_t), intent(in), target :: pn_header
      character(32), intent(in) :: &
           md5sum_process, md5sum_model, md5sum_parameters
      type(phs_parameters_t), intent(in) :: phs_par
      logical, intent(out) :: match
      type(parse_node_t), pointer :: pn_md5sum, pn_rval, pn_ival
      character(32) :: md5sum
      type(phs_parameters_t) :: phs_par_old
      pn_md5sum => parse_node_get_sub_ptr (pn_header, 3)
      md5sum = parse_node_get_string (pn_md5sum)
      if (md5sum /= "" .and. md5sum /= md5sum_process) then
         call msg_message ("Rebuilding phase space (process has changed)")
         match = .false.;  return
      end if
      pn_md5sum => parse_node_get_next_ptr (pn_md5sum, 3)
      md5sum = parse_node_get_string (pn_md5sum)
      if (md5sum /= "" .and. md5sum /= md5sum_model) then
         call msg_message ("Rebuilding phase space (model has changed)")
         match = .false.;  return
      end if
      pn_md5sum => parse_node_get_next_ptr (pn_md5sum, 3)
      md5sum = parse_node_get_string (pn_md5sum)
      if (md5sum /= "" .and. md5sum /= md5sum_parameters) then
         call msg_message &
              ("Rebuilding phase space (model parameters have changed)")
         match = .false.;  return
      end if
      pn_rval => parse_node_get_next_ptr (pn_md5sum, 3)
      phs_par_old%sqrts = parse_node_get_real (pn_rval)
      pn_rval => parse_node_get_next_ptr (pn_rval, 3)
      phs_par_old%m_threshold_s = parse_node_get_real (pn_rval)
      pn_rval => parse_node_get_next_ptr (pn_rval, 3)
      phs_par_old%m_threshold_t = parse_node_get_real (pn_rval)
      pn_ival => parse_node_get_next_ptr (pn_rval, 3)
      phs_par_old%off_shell = parse_node_get_integer (pn_ival)
      pn_ival => parse_node_get_next_ptr (pn_ival, 3)
      phs_par_old%t_channel = parse_node_get_integer (pn_ival)
      if (phs_par_old /= phs_par) then
         call msg_message &
```

```
              ("Rebuilding phase space (phase-space parameters have changed)")
          match = .false.;   return
       end if
       match = .true.
    end subroutine phs_forest_check_input
```

Initialize a specific tree in the forest, using the contents of the 'tree' node. First,
count the bincodes, allocate an array and read them in, and make the tree. Each
t-channel tree is flipped to s-channel. Then, find mappings and initialize them.

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_tree_set (tree, node, model)
      type(phs_tree_t), intent(inout) :: tree
      type(parse_node_t), intent(in), target :: node
      type(model_t), intent(in), target :: model
      type(parse_node_t), pointer :: node_bincodes, node_mapping
      integer :: n_bincodes
      integer(TC), dimension(:), allocatable :: bincode
      integer :: b, n_mappings, m
      integer(TC) :: k
      type(string_t) :: type
      integer :: pdg
      node_bincodes => parse_node_get_sub_ptr (node, 2)
      n_bincodes = parse_node_get_n_sub (node_bincodes)
      allocate (bincode (n_bincodes))
      do b = 1, n_bincodes
         bincode(b) = parse_node_get_integer &
              (parse_node_get_sub_ptr (node_bincodes, b))
      end do
      call phs_tree_from_array (tree, bincode)
      call phs_tree_flip_t_to_s_channel (tree)
      call phs_tree_canonicalize (tree)
      n_mappings = parse_node_get_n_sub (node) - 2
      do m = 1, n_mappings
         node_mapping => parse_node_get_sub_ptr (node, m+2)
         k = parse_node_get_integer &
              (parse_node_get_sub_ptr (node_mapping, 2))
         type = parse_node_get_key &
              (parse_node_get_sub_ptr (node_mapping, 3))
         pdg = parse_node_get_integer &
              (parse_node_get_sub_ptr (node_mapping, 4))
         call phs_tree_init_mapping (tree, k, type, pdg, model)
      end do
    end subroutine phs_tree_set
```

12.3.8   Preparation

The trees that we read from file do not carry flavor information. This is set
separately:

   The flavor list must be unique for a unique set of masses; if a given particle
can have different flavor, the mass must be degenerate, so we can choose one of
the possible flavor combinations.

⟨PHS forests: public⟩+≡
```
    public :: phs_forest_set_flavors
```

⟨PHS forests: procedures⟩+≡
```
    subroutine phs_forest_set_flavors (forest, flv)
      type(phs_forest_t), intent(inout) :: forest
      type(flavor_t), dimension(:), intent(in) :: flv
      allocate (forest%flv (size (flv)))
      forest%flv = flv
    end subroutine phs_forest_set_flavors
```

Once the parameter set is fixed, the masses and the widths of the particles are known and the `mass_sum` arrays as well as the mapping parameters can be computed.

⟨PHS forests: public⟩+≡
```
    public :: phs_forest_set_parameters
```

⟨PHS forests: procedures⟩+≡
```
    subroutine phs_forest_set_parameters &
        (forest, mapping_defaults, variable_limits)
      type(phs_forest_t), intent(inout) :: forest
      type(mapping_defaults_t), intent(in) :: mapping_defaults
      logical, intent(in) :: variable_limits
      integer :: g, t
      do g = 1, size (forest%grove)
         do t = 1, size (forest%grove(g)%tree)
            call phs_tree_set_mass_sum &
                (forest%grove(g)%tree(t), forest%flv(forest%n_in+1:))
            call phs_tree_set_mapping_parameters (forest%grove(g)%tree(t), &
                mapping_defaults, variable_limits)
         end do
      end do
    end subroutine phs_forest_set_parameters
```

12.3.9 Accessing the particle arrays

Set the incoming particles from the contents of an interaction.

⟨PHS forests: public⟩+≡
```
    public :: phs_forest_set_prt_in
```

⟨PHS forests: procedures⟩+≡
```
    subroutine phs_forest_set_prt_in (forest, int, lt_cm_to_lab)
      type(phs_forest_t), intent(inout) :: forest
      type(interaction_t), intent(in) :: int
      type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
      if (present (lt_cm_to_lab)) then
         call phs_prt_set_momentum (forest%prt_in, &
             inverse (lt_cm_to_lab) * &
             interaction_get_momenta (int, outgoing=.false.))
      else
         call phs_prt_set_momentum (forest%prt_in, &
             interaction_get_momenta (int, outgoing=.false.))
      end if
```

```
      call phs_prt_set_msq (forest%prt_in, &
           flavor_get_mass (forest%flv(:forest%n_in)) ** 2)
      call phs_prt_set_defined (forest%prt_in)
    end subroutine phs_forest_set_prt_in
```

Extract the outgoing particles and insert into an interaction.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_get_prt_out
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_forest_get_prt_out (forest, int, lt_cm_to_lab)
      type(phs_forest_t), intent(in) :: forest
      type(interaction_t), intent(inout) :: int
      type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
      if (present (lt_cm_to_lab)) then
         call interaction_set_momenta (int, &
              lt_cm_to_lab * &
              phs_prt_get_momentum (forest%prt_out), outgoing=.true.)
      else
         call interaction_set_momenta (int, &
              phs_prt_get_momentum (forest%prt_out), outgoing=.true.)
      end if
    end subroutine phs_forest_get_prt_out
```

12.3.10 Find equivalences among phase-space trees

Scan phase space for equivalences. We generate the complete set of unique
permutations for the given list of outgoing particles, and use this for scanning
equivalences within each grove. We scan all pairs of trees, using all permu-
tations. This implies that trivial equivalences are included, and equivalences
between different trees are recorded twice. This is intentional.

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_grove_set_equivalences (grove, perm_array)
      type(phs_grove_t), intent(inout) :: grove
      type(permutation_t), dimension(:), intent(in) :: perm_array
      type(equivalence_t), pointer :: eq
      integer :: t1, t2, i
      do t1 = 1, size (grove%tree)
         do t2 = 1, size (grove%tree)
            SCAN_PERM: do i = 1, size (perm_array)
               if (phs_tree_equivalent &
                    (grove%tree(t1), grove%tree(t2), perm_array(i))) then
                  call equivalence_list_add &
                       (grove%equivalence_list, t1, t2, perm_array(i))
                  eq => grove%equivalence_list%last
                  call phs_tree_find_msq_permutation &
                       (grove%tree(t1), grove%tree(t2), eq%perm, &
                        eq%msq_perm)
                  call phs_tree_find_angle_permutation &
                       (grove%tree(t1), grove%tree(t2), eq%perm, &
                        eq%angle_perm, eq%angle_sig)
               end if
```

```
        end do SCAN_PERM
      end do
    end do
  end subroutine phs_grove_set_equivalences
```

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_set_equivalences
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_forest_set_equivalences (forest)
      type(phs_forest_t), intent(inout) :: forest
      type(permutation_t), dimension(:), allocatable :: perm_array
      integer :: i
      call permutation_array_make &
           (perm_array, flavor_get_pdg (forest%flv(forest%n_in+1:)))
      do i = 1, size (forest%grove)
         call phs_grove_set_equivalences (forest%grove(i), perm_array)
      end do
      forest%n_equivalences = sum (forest%grove%equivalence_list%length)
    end subroutine phs_forest_set_equivalences
```

12.3.11 Interface for `VAMP` equivalences

Transform the equivalence lists into a `vamp_equivalence_list` object. The additional information that we need is: the number of extra integration dimensions (associated to structure functions), which ones of those correspond to external event generators (so that the binning must not be adapted), and whether there is any dependence on the first azimuthal angle (so its binning may be adapted or fixed).

The permutations of masses and angles are translated into permutations of integration dimensions, where the correct mapping modes are important: `msq` values are always mapped 1:1 while azimuthal angles may need an offset by $1/2$ and polar angles may need an inversion. The first azimuthal angle should not be adapted to since the dependence of the matrix element is (usually) trivial.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_setup_vamp_equivalences
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_forest_setup_vamp_equivalences &
         (forest, n_dim_extra, externally_generated, azimuthal_dependence, &
          vamp_eq)
      type(phs_forest_t), intent(in) :: forest
      integer, intent(in) :: n_dim_extra
      logical, dimension(n_dim_extra), intent(in) :: externally_generated
      logical, intent(in) :: azimuthal_dependence
      type(vamp_equivalences_t), intent(out) :: vamp_eq
      integer :: n_equivalences, n_channels, n_dim, n_masses, n_angles
      integer, dimension(forest%n_dimensions + n_dim_extra) :: perm, mode
      integer :: mode_azimuthal_angle
      type(equivalence_t), pointer :: eq
      integer :: i, j, g
      integer :: left, right
```

```
n_equivalences = forest%n_equivalences
n_channels = forest%n_trees
n_dim = forest%n_dimensions
n_masses = forest%n_masses
n_angles = forest%n_angles
call vamp_equivalences_init &
     (vamp_eq, n_equivalences, n_channels, n_dim + n_dim_extra)
if (azimuthal_dependence) then
   mode_azimuthal_angle = VEQ_IDENTITY
else
   mode_azimuthal_angle = VEQ_INVARIANT
end if
g = 0
eq => null ()
do i = 1, n_equivalences
   if (.not. associated (eq)) then
      g = g + 1
      eq => forest%grove(g)%equivalence_list%first
   end if
   do j = 1, n_masses
      perm(j) = permute (j, eq%msq_perm)
      mode(j) = VEQ_IDENTITY
   end do
   do j = 1, n_angles
      perm(n_masses+j) = n_masses + permute (j, eq%angle_perm)
      if (j == 1) then
         mode(n_masses+j) = mode_azimuthal_angle   ! first azimuthal angle
      else if (mod(j,2) == 1) then
         mode(n_masses+j) = VEQ_SYMMETRIC           ! other azimuthal angles
      else if (eq%angle_sig(j)) then
         mode(n_masses+j) = VEQ_IDENTITY            ! polar angle +
      else
         mode(n_masses+j) = VEQ_INVERT              ! polar angle -
      end if
   end do
   do  j = 1, n_dim_extra
      perm(n_dim+j) = n_dim + j
      if (externally_generated(j)) then
         mode(n_dim+j) = VEQ_INVARIANT
      else
         mode(n_dim+j) = VEQ_IDENTITY
      end if
   end do
   left = eq%left + forest%grove(g)%tree_count_offset
   right = eq%right + forest%grove(g)%tree_count_offset
   call vamp_equivalence_set (vamp_eq, i, left, right, perm, mode)
   eq => eq%next
end do
call vamp_equivalences_complete (vamp_eq)
end subroutine phs_forest_setup_vamp_equivalences
```

12.3.12 Phase-space evaluation

⟨*PHS forests: public*⟩+≡

```
public :: phs_forest_evaluate_phase_space
```

⟨*PHS forests: procedures*⟩+≡

```fortran
subroutine phs_forest_evaluate_phase_space &
     (forest, channel, active, sqrts, x, factor, volume, ok)
  type(phs_forest_t), intent(inout) :: forest
  integer, intent(in) :: channel
  logical, dimension(:), intent(in) :: active
  real(default), intent(in) :: sqrts
  real(default), dimension(:,:), intent(inout) :: x
  real(default), dimension(:), intent(out) :: factor
  real(default), intent(out) :: volume
  logical, intent(out) :: ok
  integer :: g, t, ch
  integer(TC) :: k, k_root, k_in
  g = forest%grove_lookup (channel)
  t = channel - forest%grove(g)%tree_count_offset
  call phs_prt_set_undefined (forest%prt)
  call phs_prt_set_undefined (forest%prt_out)
  k_in = forest%n_tot
  forall (k = 1:forest%n_in)
     forest%prt(ibset(0,k_in-k)) = forest%prt_in(k)
  end forall
  do k = 1, forest%n_out
     call phs_prt_set_msq (forest%prt(ibset(0,k-1)), &
          flavor_get_mass (forest%flv(forest%n_in+k)) ** 2)
  end do
  k_root = 2**forest%n_out - 1
  select case (forest%n_in)
  case (1)
     forest%prt(k_root) = forest%prt_in(1)
  case (2)
     call phs_prt_combine &
          (forest%prt(k_root), forest%prt_in(1), forest%prt_in(2))
  end select
  call phs_tree_compute_momenta_from_x (forest%grove(g)%tree(t), &
       forest%prt, factor(channel), volume, sqrts, x(:,channel), ok)
  if (ok) then
     ch = 0
     do g = 1, size (forest%grove)
        do t = 1, size (forest%grove(g)%tree)
           ch = ch + 1
           if (ch == channel)  cycle
           if (active(ch)) then
              call phs_tree_combine_particles &
                   (forest%grove(g)%tree(t), forest%prt)
              call phs_tree_compute_x_from_momenta &
                   (forest%grove(g)%tree(t), &
                    forest%prt, factor(ch), sqrts, x(:,ch))
           end if
        end do
     end do
```

```
      forall (k = 1:forest%n_out)
         forest%prt_out(k) = forest%prt(ibset(0,k-1))
      end forall
   end if
end subroutine phs_forest_evaluate_phase_space
```

12.3.13 Test of forest setup

Write a possible phase-space file for a $2 \to 3$ process and make the corresponding
forest. Print the forest and the resulting VAMP equivalence array. Choose some
in-particle momenta and a random-number array and evaluate out-particles and
phase-space factors.

⟨*PHS forests: public*⟩+≡
```
   public :: phs_forest_test
```

⟨*PHS forests: procedures*⟩+≡
```
   subroutine phs_forest_test ()
     use os_interface, only: os_data_t
     type(os_data_t) :: os_data
     type(phs_forest_t) :: forest
     type(model_t), pointer :: model
     type(string_t) :: process_id
     type(flavor_t), dimension(5) :: flv
     type(vamp_equivalences_t) :: vamp_eq
     type(string_t) :: filename
     type(interaction_t) :: int
     integer :: unit
     integer, parameter :: u = 20
     integer, parameter :: n_dim_extra = 2
     logical, dimension(2), parameter :: &
         externally_generated = (/ .true., .false. /)
     logical, parameter :: azimuthal_dependence = .false.
     type(mapping_defaults_t) :: mapping_defaults
     logical :: found_process, ok
     integer :: channel, ch
     logical, dimension(4) :: active = .true.
     real(default) :: sqrts = 1000
     real(default), dimension(5,4) :: x
     real(default), dimension(4) :: factor
     real(default) :: volume
     print *, "*** Read model file"
     call syntax_model_file_init ()
     call model_list_read_model &
         (var_str("QCD"), var_str("test.mdl"), os_data, model)
     call syntax_model_file_final ()
     print *
     print *, "*** Create phase-space file 'test.phs'"
     call flavor_init (flv, (/ 11, -11, 11, -11, 22 /), model)
     open (file="test.phs", unit=u, action="write")
     write (u, *) "process foo"
     write (u, *) "  grove"
     write (u, *) "    tree 3 7"
     write (u, *) "      map 3 s_channel 23"
```

```
          write (u, *) "    tree 5 7"
          write (u, *) "    tree 6 7"
          write (u, *) "  grove"
          write (u, *) "    tree 9 11"
          write (u, *) "      map 9 t_channel 22"
          close (u)
          print *
          print *, "*** Read phase-space file 'test.phs'"
          call syntax_phs_forest_init ()
          process_id = "foo"
          unit = free_unit ()
          filename = "test.phs"
          call phs_forest_read &
               (forest, filename, process_id, 2, 3, model, found_process)
          print *
          print *, "*** Set parameters, flavors, equiv, momenta"
          call phs_forest_set_flavors (forest, flv)
          call phs_forest_set_parameters (forest, mapping_defaults, .false.)
          call phs_forest_set_equivalences (forest)
          call interaction_init (int, 2, 0, 3)
          call interaction_set_momentum (int, &
               vector4_moving (500._default, 500._default, 3), 1)
          call interaction_set_momentum (int, &
               vector4_moving (500._default,-500._default, 3), 2)
          call phs_forest_set_prt_in (forest, int)
          channel = 2
          x = 0
          x(:,channel) = (/ 0.3, 0.4, 0.1, 0.9, 0.6 /)
1         format (5(1x,G12.5))
          print *, "Input values:"
          print 1, x(:,channel)
          print *
          print *, "*** Evaluate phase space"
          call phs_forest_evaluate_phase_space (forest, &
               channel, active, sqrts, x, factor, volume, ok)
          call phs_forest_get_prt_out (forest, int)
          print *, "Output values:"
          do ch = 1, 4
             print 1, x(:,ch)
          end do
          call interaction_write (int)
          print *, "factors:"
          print 1, factor
          print *, "volume:"
          print 1, volume
          call phs_forest_write (forest)
          print *
          print *, "*** Compute equivalences"
          call phs_forest_setup_vamp_equivalences (forest, &
               n_dim_extra, externally_generated, azimuthal_dependence, &
               vamp_eq)
          call vamp_equivalences_write (vamp_eq)
          print *
          print *, "*** Cleanup"
```

```
    call vamp_equivalences_final (vamp_eq)
    call phs_forest_final (forest)
    call syntax_phs_forest_final ()
  end subroutine phs_forest_test
```

12.4   Finding phase space parameterizations

If the phase space configuration is not found in the appropriate file, we should generate one.

The idea is to construct all Feynman diagrams subject to certain constraints which eliminate everything that is probably irrelevant for the integration. These Feynman diagrams (cascades) are grouped in groves by finding equivalence classes related by symmetry and ordered with respect to their importance (resonances). Finally, the result (or part of it) is written to file and used for the integration.

This module may eventually disappear and be replaced by CAML code. In particular, we need here a set of Feynman rules (vertices with particle codes, but not the factors). Thus, the module works for the Standard Model only.

Note that this module is stand-alone, it communicates to the main program only via the generated ASCII phase-space configuration file.

⟨cascades.f90⟩≡
  ⟨*File header*⟩

```
  module cascades
```

  ⟨*Use kinds*⟩
```
    use kinds, only: TC, i8, i32 !NODEP!
```
  ⟨*Use strings*⟩
```
    use limits, only: CASCADE_SET_FILL_RATIO, MAX_WARN_RESONANCE !NODEP!
```
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use hashes
    use pdg_arrays, only: UNDEFINED
    use models
    use flavors
    use phs_forests
```

  ⟨*Standard module head*⟩

  ⟨*Cascades: public*⟩

  ⟨*Cascades: parameters*⟩

  ⟨*Cascades: types*⟩

  ⟨*Cascades: interfaces*⟩

```
  contains
```

  ⟨*Cascades: procedures*⟩

```
end module cascades
```

12.4.1 The mapping modes

The valid mapping modes, to be used below:

⟨*Mapping modes*⟩≡
```
    integer, parameter :: &
         & NO_MAPPING = 0, S_CHANNEL = 1, T_CHANNEL =  2, U_CHANNEL = -2, &
         & RADIATION = 3, COLLINEAR = 4, INFRARED = 5
```

⟨*Cascades: parameters*⟩≡
  ⟨*Mapping modes*⟩

12.4.2 The cascade type

A cascade is essentially the same as a decay tree (both definitions may be merged in a later version). It contains a linked tree of nodes, each of which representing an internal particle. In contrast to decay trees, each node has a definite particle code. These nodes need not be modified, therefore we can use pointers and do not have to copy them. Thus, physically each cascades has only a single node, the mother particle. However, to be able to compare trees quickly, we store in addition an array of binary codes which is always sorted in ascending order. This is accompanied by a corresponding list of particle codes. The index is the location of the corresponding cascade in the cascade set, this may be used to access the daughters directly.

The real mass is the particle mass belonging to the particle code. The minimal mass is the sum of the real masses of all its daughters; this is the kinematical cutoff. The effective mass may be zero if the particle mass is below a certain threshold; it may be the real mass if the particle is resonant; or it may be some other value.

The logical `t_channel` is set if this a *t*-channel line, while `initial` is true only for an initial particle. Note that both initial particles are also `t_channel` by definition, and that they are distinguished by the direction of the tree: One of them decays and is the root of the tree, while the other one is one of the leaves.

The cascade is a list of nodes (particles) which are linked via the `daughter` entries. The node is the mother particle of the decay cascade. Much of the information in the nodes is repeated in arrays, to be accessible more easily. The arrays will be kept sorted by binary codes.

The counter `n_t_channel` is non-negative once an initial particle is included in the tree: then, it counts the number of *t*-channel lines.

The `multiplicity` is the number of branchings to follow until all daughters are on-shell. A resonant or non-decaying particle has multiplicity one. Merging nodes, the multiplicities add unless the mother is a resonance. An initial or final node has multiplicity zero.

The arrays correspond to the subnode tree `tree` of the current cascade. PDG codes are stored only for those positions which are resonant, with the exception of the last entry, i.e., the current node. Other positions, in particular external legs, are assigned undefined PDG code.

A cascade is uniquely identified by its tree, the tree of PDG codes, and the tree of mappings. The tree of resonances is kept only to mask the PDG tree as described above.

⟨*Cascades: types*⟩≡

```
type :: cascade_t
  private
  ! counters
  integer :: index = 0
  integer :: grove = 0
  ! status
  logical :: active = .false.
  logical :: complete = .false.
  logical :: incoming = .false.
  ! this node
  integer(TC) :: bincode = 0
  type(flavor_t) :: flv
  integer :: pdg = UNDEFINED
  logical :: is_vector = .false.
  real(default) :: m_min = 0
  real(default) :: m_rea = 0
  real(default) :: m_eff = 0
  integer :: mapping = NO_MAPPING
  logical :: on_shell = .false.
  logical :: resonant = .false.
  logical :: log_enhanced = .false.
  logical :: t_channel = .false.
  ! global tree properties
  integer :: multiplicity = 0
  integer :: internal = 0
  integer :: n_resonances = 0
  integer :: n_log_enhanced = 0
  integer :: n_t_channel = -1
  ! the sub-node tree
  integer :: depth = 0
  integer(TC), dimension(:), allocatable :: tree
  integer, dimension(:), allocatable :: tree_pdg
  integer, dimension(:), allocatable :: tree_mapping
  logical, dimension(:), allocatable :: tree_resonant
  ! branch connections
  logical :: has_children = .false.
  type(cascade_t), pointer :: daughter1 => null ()
  type(cascade_t), pointer :: daughter2 => null ()
  type(cascade_t), pointer :: mother => null ()
  ! next in list
  type(cascade_t), pointer :: next => null ()
end type cascade_t
```

⟨*Cascades: procedures*⟩≡

```
subroutine cascade_init (cascade, depth)
  type(cascade_t), intent(out) :: cascade
  integer, intent(in) :: depth
  integer, save :: index = 0
  index = cascade_index ()
```

```
      cascade%index = index
      cascade%depth = depth
      cascade%active = .true.
      allocate (cascade%tree (depth))
      allocate (cascade%tree_pdg (depth))
      allocate (cascade%tree_mapping (depth))
      allocate (cascade%tree_resonant (depth))
    end subroutine cascade_init
```

Keep and increment a global index

⟨*Cascades: procedures*⟩+≡

```
    function cascade_index (seed) result (index)
      integer :: index
      integer, intent(in), optional :: seed
      integer, save :: i = 0
      if (present (seed))  i = seed
      i = i + 1
      index = i
    end function cascade_index
```

We need three versions of writing cascades. This goes to the phase-space file:

⟨*Cascades: procedures*⟩+≡

```
    subroutine cascade_write_file_format (cascade, unit)
      type(cascade_t), intent(in) :: cascade
      integer, intent(in), optional :: unit
      integer :: u, i
 1    format(3x,A,1x,40(1x,I4))
 2    format(3x,A,1x,I3,1x,A,1x,I7)
      u = output_unit (unit);  if (u < 0)  return
      write (u, 1)  "tree", reduced (cascade%tree)
      do i = 1, cascade%depth
        select case (cascade%tree_mapping(i))
        case (NO_MAPPING)
        case (S_CHANNEL)
          write(u,2) 'map', &
              cascade%tree(i), 's_channel', abs (cascade%tree_pdg(i))
        case (T_CHANNEL)
          write(u,2) 'map', &
              cascade%tree(i), 't_channel', abs (cascade%tree_pdg(i))
        case (U_CHANNEL)
          write(u,2) 'map', &
              cascade%tree(i), 'u_channel', abs (cascade%tree_pdg(i))
        case (RADIATION)
          write(u,2) 'map', &
              cascade%tree(i), 'radiation', abs (cascade%tree_pdg(i))
        case (COLLINEAR)
          write(u,2) 'map', &
              cascade%tree(i), 'collinear', abs (cascade%tree_pdg(i))
        case (INFRARED)
          write(u,2) 'map', &
              cascade%tree(i), 'infrared',  abs (cascade%tree_pdg(i))
        case default
          call msg_bug (" Impossible mapping mode encountered")
        end select
```

```
        end do
    contains
      function reduced (array)
        integer(TC), dimension(:), intent(in) :: array
        integer(TC), dimension(max((size(array)-3)/2, 1)) :: reduced
        integer :: i, j
        j = 1
        do i=1, size(array)
          if (decay_level(array(i)) > 1) then
            reduced(j) = array(i)
            j = j+1
          end if
        end do
      end function reduced
      function decay_level (k) result (l)
        integer(TC), intent(in) :: k
        integer :: l
        integer :: i
        l = 0
        do i = 0, bit_size(k) - 1
          if (btest(k,i)) l = l + 1
        end do
      end function decay_level
      subroutine start_comment (u)
        integer, intent(in) :: u
        write(u, '(1x,A)', advance='no') '!'
      end subroutine start_comment
    end subroutine cascade_write_file_format
```

This creates metapost source for graphical display:

⟨*Cascades: procedures*⟩+≡

```
    subroutine cascade_write_graph_format (cascade, count, unit)
      type(cascade_t), intent(in) :: cascade
      integer, intent(in) :: count
      integer, intent(in), optional :: unit
      integer :: u
      integer(TC) :: mask
      type(string_t) :: left_str, right_str
      u = output_unit (unit);  if (u < 0)  return
      mask = 2**((cascade%depth+3)/2) - 1
      left_str = ""
      right_str = ""
      write (u, '(A)') "\begin{minipage}{105pt}"
      write (u, '(A)') "\vspace{30pt}"
      write (u, '(A)') "\begin{center}"
      write (u, '(A)') "\begin{fmfgraph*}(55,55)"
      call graph_write (cascade, mask)
      write (u, '(A)') "\fmfleft{" // char (extract (left_str, 2)) // "}"
      write (u, '(A)') "\fmfright{" // char (extract (right_str, 2)) // "}"
      write (u, '(A)') "\end{fmfgraph*}\\"
      write (u, '(A,I5,A)') "\fbox{$", count, "$}"
      write (u, '(A)') "\end{center}"
      write (u, '(A)') "\end{minipage}"
      write (u, '(A)') "%"
```

```
contains
  recursive subroutine graph_write (cascade, mask, reverse)
    type(cascade_t), intent(in) :: cascade
    integer(TC), intent(in) :: mask
    logical, intent(in), optional :: reverse
    logical :: rev
    rev = .false.;  if (present(reverse))  rev = reverse
    if (cascade%has_children) then
       if (.not.rev) then
          call vertex_write (cascade, cascade%daughter1, mask)
          call vertex_write (cascade, cascade%daughter2, mask)
       else
          call vertex_write (cascade, cascade%daughter2, mask, .true.)
          call vertex_write (cascade, cascade%daughter1, mask, .true.)
       end if
       if (cascade%complete) then
          call vertex_write (cascade, cascade%mother, mask, .true.)
          write (u, '(A,I3,A)') &
               "\fmfv{d.shape=square}{v", cascade%bincode, "}"
       end if
    else
       if (cascade%incoming) then
          call external_write (cascade%bincode, &
               flavor_get_tex_name (flavor_anti (cascade%flv)), left_str)
       else
          call external_write (cascade%bincode, &
               flavor_get_tex_name (cascade%flv), right_str)
       end if
    end if
  end subroutine graph_write
  recursive subroutine vertex_write (cascade, daughter, mask, reverse)
    type(cascade_t), intent(in) :: cascade, daughter
    integer(TC), intent(in) :: mask
    logical, intent(in), optional :: reverse
    call graph_write (daughter, mask, reverse)
    if (daughter%has_children) then
       call line_write (cascade%bincode, daughter%bincode, cascade%flv, &
            mapping=daughter%mapping)
    else
       call line_write (cascade%bincode, daughter%bincode, cascade%flv)
    end if
  end subroutine vertex_write
  subroutine line_write (i1, i2, flv, mapping)
    integer(TC), intent(in) :: i1, i2
    type(flavor_t), intent(in) :: flv
    integer, intent(in), optional :: mapping
    integer :: k1, k2
    type(string_t) :: prt_type
    select case (flavor_get_spin_type (flv))
    case (SCALAR);       prt_type = "plain"
    case (SPINOR);       prt_type = "fermion"
    case (VECTOR);       prt_type = "boson"
    case (VECTORSPINOR); prt_type = "fermion"
    case (TENSOR);       prt_type = "dbl_wiggly"
```

```
            case default;        prt_type = "dashes"
         end select
         if (flavor_is_antiparticle (flv)) then
            k1 = i2;  k2 = i1
         else
            k1 = i1;  k2 = i2
         end if
         if (present (mapping)) then
            select case (mapping)
            case (S_CHANNEL)
               write (u, '(A,I3,A,I3,A)') "\fmf{" // char (prt_type) // &
                    & ",f=blue,lab=\sm\blue$" // &
                    & char (flavor_get_tex_name (flv)) // "$}" // &
                    & "{v", k1, ",v", k2, "}"
            case (T_CHANNEL, U_CHANNEL)
               write (u, '(A,I3,A,I3,A)') "\fmf{" // char (prt_type) // &
                    & ",f=cyan,lab=\sm\cyan$" // &
                    & char (flavor_get_tex_name (flv)) // "$}" // &
                    & "{v", k1, ",v", k2, "}"
            case (RADIATION)
               write (u, '(A,I3,A,I3,A)') "\fmf{" // char (prt_type) // &
                    & ",f=green,lab=\sm\green$" // &
                    & char (flavor_get_tex_name (flv)) // "$}" // &
                    & "{v", k1, ",v", k2, "}"
            case (COLLINEAR)
               write (u, '(A,I3,A,I3,A)') "\fmf{" // char (prt_type) // &
                    & ",f=magenta,lab=\sm\magenta$" // &
                    & char (flavor_get_tex_name (flv)) // "$}" // &
                    & "{v", k1, ",v", k2, "}"
            case (INFRARED)
               write (u, '(A,I3,A,I3,A)') "\fmf{" // char (prt_type) // &
                    & ",f=red,lab=\sm\red$" // &
                    & char (flavor_get_tex_name (flv)) // "$}" // &
                    & "{v", k1, ",v", k2, "}"
            case default
               write (u, '(A,I3,A,I3,A)') "\fmf{" // char (prt_type) // &
                    & ",f=black}" // &
                    & "{v", k1, ",v", k2, "}"
            end select
         else
            write (u, '(A,I3,A,I3,A)') "\fmf{" // char (prt_type) // &
                    & "}" // &
                    & "{v", k1, ",v", k2, "}"
         end if
       end subroutine line_write
       subroutine external_write (bincode, name, ext_str)
         integer(TC), intent(in) :: bincode
         type(string_t), intent(in) :: name
         type(string_t), intent(inout) :: ext_str
         character(len=5) :: str
         write (str, '(A2,I3)') ",v", bincode
         ext_str = ext_str // str
         write (u, '(A,I3,A,I3,A)') "\fmflabel{\sm$" &
            // char (name) &
```

```
                // "\,(", bincode, ")" &
                // "$}{v", bincode, "}"
          end subroutine external_write
       end subroutine cascade_write_graph_format
```

This is for screen/debugging output:

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_write (cascade, unit)
      type(cascade_t), intent(in) :: cascade
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, *) 'Cascade #', cascade%index
      write (u, *) ' Grove:        #', cascade%grove
      write (u, *) ' act/cmp/inc: ', &
           cascade%active, cascade%complete, cascade%incoming
      write (u, *) ' Bincode:       ', cascade%bincode
      write (u, "(1x,A)", advance="no") ' Flavor:        '
      call flavor_write (cascade%flv, unit)
      write (u, *) ' Active flavor:', cascade%pdg
      write (u, *) ' Is vector:    ', cascade%is_vector
      write (u, *) ' Mass (m/r/e): ', &
           cascade%m_min, cascade%m_rea, cascade%m_eff
      write (u, *) ' Mapping:      ', cascade%mapping
      write (u, *) ' res/log/tch: ', &
           cascade%resonant, cascade%log_enhanced, cascade%t_channel
      write (u, *) ' Multiplicity: ', cascade%multiplicity
      write (u, *) ' n internal:   ', cascade%internal
      write (u, *) ' n res/log/tch:', &
           cascade%n_resonances, cascade%n_log_enhanced, cascade%n_t_channel
      write (u, *) ' Depth:        ', cascade%depth
      write (u, *) ' Tree:         ', cascade%tree
      write (u, *) ' Tree(PDG):    ', cascade%tree_pdg
      write (u, *) ' Tree(mapping):', cascade%tree_mapping
      write (u, *) ' Tree(res):    ', cascade%tree_resonant
      if (cascade%has_children) then
         write (u, *) ' Daughter1/2: ', &
              cascade%daughter1%index, cascade%daughter2%index
      end if
      if (associated (cascade%mother)) then
         write (u, *) ' Mother:       ', cascade%mother%index
      end if
    end subroutine cascade_write
```

12.4.3   Creating new cascades

This initializes a single-particle cascade (external, final state). The PDG entry
in the tree is set undefined because the cascade is not resonant. However, the
flavor entry is set, so the cascade flavor is identified nevertheless.

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_init_outgoing (cascade, flv, pos, m_thr)
      type(cascade_t), intent(out) :: cascade
```

```
      type(flavor_t), intent(in) :: flv
      integer, intent(in) :: pos
      real(default), intent(in) :: m_thr
      call cascade_init (cascade, 1)
      cascade%bincode = ibset (0_TC, pos-1)
      cascade%flv = flv
      cascade%pdg = 0
      cascade%is_vector = flavor_get_spin_type (flv) == VECTOR
      cascade%m_min = flavor_get_mass (flv)
      cascade%m_rea = cascade%m_min
      if (cascade%m_rea >= m_thr) then
         cascade%m_eff = cascade%m_rea
      end if
      cascade%on_shell = .true.
      cascade%multiplicity = 1
      cascade%tree(1) = cascade%bincode
      cascade%tree_pdg(1) = cascade%pdg
      cascade%tree_mapping(1) = NO_MAPPING
      cascade%tree_resonant(1) = .false.
    end subroutine cascade_init_outgoing
```

The same for an incoming line:

⟨*Cascades: procedures*⟩+≡

```
    subroutine cascade_init_incoming (cascade, flv, pos, m_thr)
      type(cascade_t), intent(out) :: cascade
      type(flavor_t), intent(in) :: flv
      integer, intent(in) :: pos
      real(default), intent(in) :: m_thr
      call cascade_init (cascade, 1)
      cascade%incoming = .true.
      cascade%bincode = ibset (0_TC, pos-1)
      cascade%flv = flavor_anti (flv)
      cascade%pdg = 0
      cascade%is_vector = flavor_get_spin_type (flv) == VECTOR
      cascade%m_min = flavor_get_mass (flv)
      cascade%m_rea = cascade%m_min
      if (cascade%m_rea >= m_thr) then
         cascade%m_eff = cascade%m_rea
      end if
      cascade%on_shell = .true.
      cascade%n_t_channel = 0
      cascade%tree(1) = cascade%bincode
      cascade%tree_pdg(1) = cascade%pdg
      cascade%tree_mapping(1) = NO_MAPPING
      cascade%tree_resonant(1) = .false.
    end subroutine cascade_init_incoming
```

12.4.4 Tools

This function returns true if the two cascades share no common external particle.
This is a requirement for joining them.

⟨*Cascades: interfaces*⟩≡

```
      interface operator(.disjunct.)
        module procedure cascade_disjunct
      end interface
```

⟨*Cascades: procedures*⟩+≡
```
      function cascade_disjunct (cascade1, cascade2) result (flag)
        logical :: flag
        type(cascade_t), intent(in) :: cascade1, cascade2
        flag = iand (cascade1%bincode, cascade2%bincode) == 0
      end function cascade_disjunct
```

12.4.5 Hash entries for cascades

We will set up a hash array which contains keys of and pointers to cascades.
We hold a list of cascade (pointers) within each bucket. This is not for collision
resolution, but for keeping similar, but unequal cascades together.

⟨*Cascades: types*⟩+≡
```
      type :: cascade_p
        type(cascade_t), pointer :: cascade => null ()
        type(cascade_p), pointer :: next => null ()
      end type cascade_p
```

Here is the bucket or hash entry type:

⟨*Cascades: types*⟩+≡
```
      type :: hash_entry_t
        integer(i32) :: hashval = 0
        integer(i8), dimension(:), allocatable :: key
        type(cascade_p), pointer :: first => null ()
        type(cascade_p), pointer :: last => null ()
      end type hash_entry_t
```

Finalize: just deallocate the list; the contents are just pointers.

⟨*Cascades: procedures*⟩+≡
```
      subroutine hash_entry_final (hash_entry)
        type(hash_entry_t), intent(inout) :: hash_entry
        type(cascade_p), pointer :: current
        do while (associated (hash_entry%first))
           current => hash_entry%first
           hash_entry%first => current%next
           deallocate (current)
        end do
      end subroutine hash_entry_final
```

Output: concise format for debugging, just list cascade indices.

⟨*Cascades: procedures*⟩+≡
```
      subroutine hash_entry_write (hash_entry, unit)
        type(hash_entry_t), intent(in) :: hash_entry
        integer, intent(in), optional :: unit
        type(cascade_p), pointer :: current
        integer :: u, i
```

```
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,A)", advance="no")  "Entry:"
      do i = 1, size (hash_entry%key)
         write (u, "(1x,I3)", advance="no")  hash_entry%key(i)
      end do
      write (u, "(1x,A)", advance="no")  "->"
      current => hash_entry%first
      do while (associated (current))
         write (u, "(1x,I7)", advance="no") current%cascade%index
         current => current%next
      end do
      write (u, *)
   end subroutine hash_entry_write
```

This function adds a cascade pointer to the bucket. If `ok` is present, check first
if it is already there and return failure if yes.

⟨*Cascades: procedures*⟩+≡
```
   subroutine hash_entry_add_cascade_ptr (hash_entry, cascade, ok)
     type(hash_entry_t), intent(inout) :: hash_entry
     type(cascade_t), intent(in), target :: cascade
     logical, intent(out), optional :: ok
     type(cascade_p), pointer :: current
     if (present (ok)) then
        ok = .not. hash_entry_contains (hash_entry, cascade)
        if (.not. ok)  return
     end if
     allocate (current)
     current%cascade => cascade
     if (associated (hash_entry%last)) then
        hash_entry%last%next => current
     else
        hash_entry%first => current
     end if
     hash_entry%last => current
   end subroutine hash_entry_add_cascade_ptr
```

This function checks whether a cascade is already in the bucket. For incomplete
cascades, we look for an exact match. It should suffice to verify the tree, the
PDG codes, and the mapping modes. This is the information that is written to
the phase space file.

For complete cascades, we ignore the PDG code at positions with mappings
infrared, collinear, or t/u-channel. Thus a cascade which is distinguished only by
PDG code at such places, is flagged existent. If the convention is followed that
light particles come before heavier ones (in the model definition), this ensures
that the lightest particle is kept in the appropriate place, corresponding to the
strongest peak.

For external cascades (incoming/outgoing) we take the PDG code into ac-
count even though it is zeroed in the PDG-code tree.

⟨*Cascades: procedures*⟩+≡
```
   function hash_entry_contains (hash_entry, cascade) result (flag)
     logical :: flag
     type(hash_entry_t), intent(in), target :: hash_entry
```

```
type(cascade_t), intent(in) :: cascade
type(cascade_p), pointer :: current
integer, dimension(:), allocatable :: tree_pdg
flag = .false.
allocate (tree_pdg (size (cascade%tree_pdg)))
if (cascade%has_children) then
   if (cascade%complete) then
      where (cascade%tree_mapping == INFRARED .or. &
             cascade%tree_mapping == COLLINEAR .or. &
             cascade%tree_mapping == T_CHANNEL .or. &
             cascade%tree_mapping == U_CHANNEL)
         tree_pdg = 0
      elsewhere
         tree_pdg = cascade%tree_pdg
      end where
   else
      tree_pdg = cascade%tree_pdg
   end if
else
   tree_pdg = flavor_get_pdg (cascade%flv)
end if
current => hash_entry%first
do while (associated (current))
   if (current%cascade%depth == cascade%depth) then
      if (all (current%cascade%tree == cascade%tree)) then
         if (all (current%cascade%tree_mapping == cascade%tree_mapping)) &
               then
            if (all (current%cascade%tree_pdg .match. tree_pdg)) then
               flag = .true.;  return
            end if
         end if
      end if
   end if
   current => current%next
end do
end function hash_entry_contains
```

This function returns a pointer to the list entry of a cascade that is similar to the input, i.e. tree and PDG codes match but the mappings may be distributed differently.

⟨*Cascades: procedures*⟩+≡

```
function hash_entry_get_similar_p (hash_entry, cascade) result (current)
  type(cascade_p), pointer :: current
  type(hash_entry_t), intent(in), target :: hash_entry
  type(cascade_t), intent(in) :: cascade
  current => hash_entry%first
  do while (associated (current))
     if (current%cascade%depth == cascade%depth) then
        if (all (current%cascade%tree == cascade%tree)) then
           if (all (current%cascade%tree_pdg &
                 .match. cascade%tree_pdg))  return
        end if
     end if
```

```
        current => current%next
    end do
  end function hash_entry_get_similar_p
```

For PDG codes, we specify that the undefined code matches any code. This is
already defined for flavor objects, but here we need it for the codes themselves.

⟨*Cascades: interfaces*⟩+≡
```
    interface operator(.match.)
       module procedure pdg_match
    end interface
```

⟨*Cascades: procedures*⟩+≡
```
    elemental function pdg_match (pdg1, pdg2) result (flag)
      logical :: flag
      integer(TC), intent(in) :: pdg1, pdg2
      select case (pdg1)
      case (0)
         flag = .true.
      case default
         select case (pdg2)
         case (0)
            flag = .true.
         case default
            flag = pdg1 == pdg2
         end select
      end select
    end function pdg_match
```

12.4.6 The cascade set

The cascade set will later be transformed into the decay forest. It is set up as a
linked list. In addition to the usual `first` and `last` pointers, there is a `first_t`
pointer which points to the first t-channel cascade (after all s-channel cascades),
and a `first_k` pointer which points to the first final cascade (with a keystone).

As an auxiliary device, the object contains a hash array with associated
parameters where an additional pointer is stored for each cascade. The keys are
made from the relevant cascade data. This hash is used for fast detection (and
thus avoidance) of double entries in the cascade list.

⟨*Cascades: public*⟩≡
```
    public :: cascade_set_t
```

⟨*Cascades: types*⟩+≡
```
    type :: cascade_set_t
      private
      type(model_t), pointer :: model
      integer :: n_in, n_out, n_tot
      integer :: depth_out, depth_tot
      real(default) :: sqrts = 0
      real(default) :: m_threshold_s = 0
      real(default) :: m_threshold_t = 0
      integer :: off_shell = 0
      integer :: t_channel = 0
```

```
    integer :: n_groves = 0
    ! The cascade list
    type(cascade_t), pointer :: first => null ()
    type(cascade_t), pointer :: last => null ()
    type(cascade_t), pointer :: first_t => null ()
    type(cascade_t), pointer :: first_k => null ()
    ! The hashtable
    integer :: n_entries = 0
    real :: fill_ratio = 0
    integer :: n_entries_max = 0
    integer(i32) :: mask = 0
    logical :: fatal_beam_decay = .true.
    type(hash_entry_t), dimension(:), allocatable :: entry
  end type cascade_set_t
```

Return true if there are cascades which are active and complete, so the phase space file would be nonempty.

⟨*Cascades: public*⟩+≡
```
    public :: cascade_set_is_valid
```

⟨*Cascades: procedures*⟩+≡
```
    function cascade_set_is_valid (cascade_set) result (flag)
      logical :: flag
      type(cascade_set_t), intent(in) :: cascade_set
      type(cascade_t), pointer :: cascade
      flag = .false.
      cascade => cascade_set%first_k
      do while (associated (cascade))
         if (cascade%active .and. cascade%complete) then
            flag = .true.
            return
         end if
         cascade => cascade%next
      end do
    end function cascade_set_is_valid
```

The initializer sets up the hash table with some initial size guessed by looking at the number of external particles. We choose 256 for 3 external particles and a factor of 4 for each additional particle, limited at 2^{30}=1G.

⟨*Limits: public parameters*⟩+≡
```
    real, parameter, public :: CASCADE_SET_FILL_RATIO = 0.1
```

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_init (cascade_set, model, n_in, n_out, phs_par, &
         fatal_beam_decay)
      type(cascade_set_t), intent(out) :: cascade_set
      type(model_t), intent(in), target :: model
      integer, intent(in) :: n_in, n_out
      type(phs_parameters_t), intent(in) :: phs_par
      logical, intent(in) :: fatal_beam_decay
      integer :: size_guess
      cascade_set%model => model
      cascade_set%n_in = n_in
      cascade_set%n_out = n_out
```

```
      cascade_set%n_tot = n_in + n_out
      select case (n_in)
      case (1);  cascade_set%depth_out = 2 * n_out - 3
      case (2);  cascade_set%depth_out = 2 * n_out - 1
      end select
      cascade_set%depth_tot = 2 * cascade_set%n_tot - 3
      cascade_set%sqrts = phs_par%sqrts
      cascade_set%m_threshold_s = phs_par%m_threshold_s
      cascade_set%m_threshold_t = phs_par%m_threshold_t
      cascade_set%off_shell = phs_par%off_shell
      cascade_set%t_channel = phs_par%t_channel
      cascade_set%fill_ratio = CASCADE_SET_FILL_RATIO
      size_guess = ishft (256, min (2 * (cascade_set%n_tot - 3), 22))
      cascade_set%n_entries_max = size_guess * cascade_set%fill_ratio
      cascade_set%mask = size_guess - 1
      allocate (cascade_set%entry (0:cascade_set%mask))
      cascade_set%fatal_beam_decay = fatal_beam_decay
    end subroutine cascade_set_init
```

The finalizer has to delete both the hash and the list. We assume that the hash only contains pointers, so it is simply deallocated, while the list entries are physically deleted.

⟨*Cascades: public*⟩+≡
```
    public :: cascade_set_final
```

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_final (cascade_set)
      type(cascade_set_t), intent(inout), target :: cascade_set
      type(cascade_t), pointer :: current
      deallocate (cascade_set%entry)
      do while (associated (cascade_set%first))
         current => cascade_set%first
         cascade_set%first => cascade_set%first%next
         deallocate (current)
      end do
    end subroutine cascade_set_final
```

Three output routines: phase-space file, graph source code, and screen output.

This version generates the phase space file. It deals only with complete cascades.

⟨*Cascades: public*⟩+≡
```
    public :: cascade_set_write_file_format
```

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_write_file_format (cascade_set, unit)
      type(cascade_set_t), intent(in), target :: cascade_set
      integer, intent(in), optional :: unit
      type(cascade_t), pointer :: cascade
      integer :: u, grove, count
      logical :: first_in_grove
      u = output_unit (unit);  if (u < 0)  return
      count = 0
      do grove = 1, cascade_set%n_groves
```

```
            first_in_grove = .true.
            cascade => cascade_set%first_k
            do while (associated (cascade))
                if (cascade%active .and. cascade%complete) then
                    if (cascade%grove == grove) then
                        if (first_in_grove) then
                            first_in_grove = .false.
                            write (u, *)
                            write (u, "(1x,'!',1x,A,I2,A)", advance='no') &
                                'Multiplicity =', cascade%multiplicity, ","
                            select case (cascade%n_resonances)
                            case (0)
                                write (u, '(1x,A)', advance='no') 'no resonances, '
                            case (1)
                                write (u, '(1x,A)', advance='no') ' 1 resonance,  '
                            case default
                                write (u, '(1x,I2,1x,A)', advance='no') &
                                    cascade%n_resonances, 'resonances, '
                            end select
                            write (u, '(1x,I2,1x,A)', advance='no') &
                                cascade%n_log_enhanced, 'logs, '
                            select case (cascade%n_t_channel)
                            case (0);  write (u, '(1x,A)') 's-channel graph'
                            case (1);  write (u, '(1x,A)') ' 1 t-channel line'
                            case default
                                write(u,'(1x,I2,1x,A)') &
                                    cascade%n_t_channel, 't-channel lines'
                            end select
                            write (u, '(1x,A,I3)') 'grove #', grove
                        end if
                        count = count + 1
                        write (u, "(1x,'!',1x,A,1x)", advance="no")  "Channel #"
                        write (u, *)  count
                        call cascade_write_file_format (cascade, u)
                    end if
                end if
                cascade => cascade%next
            end do
        end do
    end subroutine cascade_set_write_file_format
```

This is the graph output format., the driver-file

⟨*Cascades: procedures*⟩+≡

```
    subroutine cascade_set_write_graph_format (cascade_set, filename, unit)
      type(cascade_set_t), intent(in), target :: cascade_set
      type(string_t), intent(in) :: filename
      integer, intent(in), optional :: unit
      type(cascade_t), pointer :: cascade
      integer :: u, grove, count, pgcount
      logical :: first_in_grove
      u = output_unit (unit);  if (u < 0)  return
      write (u, '(A)') "\documentclass[10pt]{article}"
      write (u, '(A)') "\usepackage{feynmp}"
```

```
            write (u, '(A)') "\usepackage{color}"
            write (u, *)
            write (u, '(A)') "\textwidth 18.5cm"
            write (u, '(A)') "\evensidemargin -1.5cm"
            write (u, '(A)') "\oddsidemargin -1.5cm"
            write (u, *)
            write (u, '(A)') "\newcommand{\blue}{\color{blue}}"
            write (u, '(A)') "\newcommand{\green}{\color{green}}"
            write (u, '(A)') "\newcommand{\red}{\color{red}}"
            write (u, '(A)') "\newcommand{\magenta}{\color{magenta}}"
            write (u, '(A)') "\newcommand{\cyan}{\color{cyan}}"
            write (u, '(A)') "\newcommand{\sm}{\footnotesize}"
            write (u, '(A)') "\setlength{\parindent}{0pt}"
            write (u, '(A)') "\setlength{\parsep}{20pt}"
            write (u, *)
            write (u, '(A)') "\begin{document}"
            write (u, '(A)') "\begin{fmffile}{" // char (filename) // "}"
            write (u, '(A)') "\fmfcmd{color magenta; magenta = red + blue;}"
            write (u, '(A)') "\fmfcmd{color cyan; cyan = green + blue;}"
            write (u, '(A)') "\begin{fmfshrink}{0.5}"
            write (u, '(A)') "\begin{flushleft}"
            write (u, *)
            write (u, '(A)') "\noindent" // &
                 & "\textbf{\large\texttt{WHIZARD} phase space channels}" // &
                 & "\hfill\today"
            write (u, *)
            write (u, '(A)') "\vspace{10pt}"
!           call write_process_tex_form (u, process_id, code, n_in, n_out)
            write (u, *)
            write (u, '(A)') "\textbf{Color code:} " // &
                 & "{\blue resonance,} " // &
                 & "{\cyan t-channel,} " // &
                 & "{\green radiation,} " // &
                 & "{\red infrared,} " // &
                 & "{\magenta collinear,} " // &
                 & "external/off-shell"
            write (u, *)
            write (u, '(A)') "\vspace{-20pt}"
            count = 0
            pgcount = 0
            do grove = 1, cascade_set%n_groves
               first_in_grove = .true.
               cascade => cascade_set%first
               do while (associated (cascade))
                  if (cascade%active .and. cascade%complete) then
                     if (cascade%grove == grove) then
                        if (first_in_grove) then
                           first_in_grove = .false.
                           write (u, *)
                           write (u, '(A)') "\vspace{20pt}"
                           write (u, '(A)') "\begin{tabular}{l}"
                           write (u, '(A,I5,A)') &
                                & "\fbox{\bf Grove \boldmath$", grove, "$} \\[10pt]"
                           write (u, '(A,I1,A)') "Multiplicity: ", &
```

```
                          cascade%multiplicity, "\\"
                 write (u, '(A,I1,A)') "Resonances:    ", &
                          cascade%n_resonances, "\\"
                 write (u, '(A,I1,A)') "Log-enhanced: ", &
                          cascade%n_log_enhanced, "\\"
                 write (u, '(A,I1,A)') "t-channel:     ", &
                          cascade%n_t_channel, ""
                 write (u, '(A)') "\end{tabular}"
              end if
              count = count + 1
              call cascade_write_graph_format (cascade, count, unit)
              if (pgcount >= 250) then
                 write (u, '(A)') "\clearpage"
                 pgcount = 0
              end if
           end if
        end if
        cascade => cascade%next
     end do
   end do
   write (u, '(A)') "\end{flushleft}"
   write (u, '(A)') "\end{fmfshrink}"
   write (u, '(A)') "\end{fmffile}"
   write (u, '(A)') "\end{document}"
  end subroutine cascade_set_write_graph_format
```

This is for screen output and debugging:

⟨*Cascades: public*⟩+≡
```
    public :: cascade_set_write
```

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_write (cascade_set, unit, active_only, complete_only)
      type(cascade_set_t), intent(in), target :: cascade_set
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: active_only, complete_only
      logical :: active, complete
      type(cascade_t), pointer :: cascade
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      active = .true.;  if (present (active_only))  active = active_only
      complete = .false.;  if (present (complete_only))  complete = complete_only
      write (u, *) "Cascade set:"
      write (u, "(3x,A)", advance="no")  "Model:"
      if (associated (cascade_set%model)) then
         write (u, "(1x,A)") char (model_get_name (cascade_set%model))
      else
         write (u, "(1x,A)") "[none]"
      end if
      write (u, "(3x,A)", advance="no")  "n_in/out/tot  ="
      write (u, *)  cascade_set%n_in, cascade_set%n_out, cascade_set%n_tot
      write (u, "(3x,A)", advance="no")  "depth_out/tot ="
      write (u, *)  cascade_set%depth_out, cascade_set%depth_tot
      write (u, "(3x,A)", advance="no")  "mass thr(s/t) ="
      write (u, *)  cascade_set%m_threshold_s, cascade_set%m_threshold_t
```

```
write (u, "(3x,A)", advance="no")  "off shell    ="
write (u, *)  cascade_set%off_shell
write (u, "(3x,A)", advance="no")  "n_groves     ="
write (u, *)  cascade_set%n_groves
write (u, *)
write (u, *) "Cascade list:"
if (associated (cascade_set%first)) then
   cascade => cascade_set%first
   do while (associated (cascade))
      if (active .and. .not. cascade%active)  cycle
      if (complete .and. .not. cascade%complete)  cycle
      call cascade_write (cascade, unit)
      cascade => cascade%next
   end do
else
   write (u, *) "[empty]"
end if
write (u, *) "Hash array"
write (u, "(3x,A)", advance="no")  "n_entries    ="
write (u, *)  cascade_set%n_entries
write (u, "(3x,A)", advance="no")  "fill_ratio   ="
write (u, *)  cascade_set%fill_ratio
write (u, "(3x,A)", advance="no")  "n_entries_max ="
write (u, *)  cascade_set%n_entries_max
write (u, "(3x,A)", advance="no")  "mask         ="
write (u, *)  cascade_set%mask
do i = 0, ubound (cascade_set%entry, 1)
   if (allocated (cascade_set%entry(i)%key)) then
      write (u, *) i
      call hash_entry_write (cascade_set%entry(i), u)
   end if
end do
end subroutine cascade_set_write
```

12.4.7   Adding cascades

Add a cascade to the set. We first try to insert it in the hash array. If successful, add it to the list. Failure indicates that it is already present, and we drop it.

The hash key is built solely from the tree array, so neither particle codes nor resonances count, just topology.

Technically, hash and list receive only pointers, so the cascade can be considered as being in either of both. We treat it as part of the list.

⟨*Cascades: procedures*⟩+≡
```
subroutine cascade_set_add (cascade_set, cascade, ok)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(cascade_t), intent(in), target :: cascade
  logical, intent(out) :: ok
  integer(i8), dimension(1) :: mold
  call cascade_set_hash_insert &
       (cascade_set, transfer (cascade%tree, mold), cascade, ok)
  if (ok)  call cascade_set_list_add (cascade_set, cascade)
end subroutine cascade_set_add
```

Add a new cascade to the list:

⟨*Cascades: procedures*⟩+≡
```
subroutine cascade_set_list_add (cascade_set, cascade)
  type(cascade_set_t), intent(inout) :: cascade_set
  type(cascade_t), intent(in), target :: cascade
  if (associated (cascade_set%last)) then
     cascade_set%last%next => cascade
  else
     cascade_set%first => cascade
  end if
  cascade_set%last => cascade
end subroutine cascade_set_list_add
```

Add a cascade entry to the hash array:

⟨*Cascades: procedures*⟩+≡
```
subroutine cascade_set_hash_insert (cascade_set, key, cascade, ok)
  type(cascade_set_t), intent(inout), target :: cascade_set
  integer(i8), dimension(:), intent(in) :: key
  type(cascade_t), intent(in), target :: cascade
  logical, intent(out) :: ok
  integer(i32) :: h
  if (cascade_set%n_entries >= cascade_set%n_entries_max) &
       call cascade_set_hash_expand (cascade_set)
  h = hash (key)
  call cascade_set_hash_insert_rec (cascade_set, h, h, key, cascade, ok)
end subroutine cascade_set_hash_insert
```

Double the hashtable size when necesssary:

⟨*Cascades: procedures*⟩+≡
```
subroutine cascade_set_hash_expand (cascade_set)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(hash_entry_t), dimension(:), allocatable, target :: table_tmp
  type(cascade_p), pointer :: current
  integer :: i, s
  allocate (table_tmp (0:cascade_set%mask))
  table_tmp = cascade_set%entry
  deallocate (cascade_set%entry)
  s = 2 * size (table_tmp)
  cascade_set%n_entries = 0
  cascade_set%n_entries_max = s * cascade_set%fill_ratio
  cascade_set%mask = s - 1
  allocate (cascade_set%entry (0:cascade_set%mask))
  do i = 0, ubound (table_tmp, 1)
     current => table_tmp(i)%first
     do while (associated (current))
        call cascade_set_hash_insert_rec &
             (cascade_set, table_tmp(i)%hashval, table_tmp(i)%hashval, &
              table_tmp(i)%key, current%cascade)
        current => current%next
     end do
  end do
```

```
        end subroutine cascade_set_hash_expand
```

Insert the cascade at the bucket determined by the hash value. If the bucket is filled, check first for a collision (unequal keys). In that case, choose the following bucket and repeat. Otherwise, add the cascade to the bucket.

If the bucket is empty, record the hash value, allocate and store the key, and then add the cascade to the bucket.

If ok is present, before insertion we check whether the cascade is already stored, and return failure if yes.

⟨*Cascades: procedures*⟩+≡
```
    recursive subroutine cascade_set_hash_insert_rec &
        (cascade_set, h, hashval, key, cascade, ok)
      type(cascade_set_t), intent(inout) :: cascade_set
      integer(i32), intent(in) :: h, hashval
      integer(i8), dimension(:), intent(in) :: key
      type(cascade_t), intent(in), target :: cascade
      logical, intent(out), optional :: ok
      integer(i32) :: i
      i = iand (h, cascade_set%mask)
      if (allocated (cascade_set%entry(i)%key)) then
         if (size (cascade_set%entry(i)%key) /= size (key)) then
            call cascade_set_hash_insert_rec &
                 (cascade_set, h + 1, hashval, key, cascade, ok)
         else if (any (cascade_set%entry(i)%key /= key)) then
            call cascade_set_hash_insert_rec &
                 (cascade_set, h + 1, hashval, key, cascade, ok)
         else
            call hash_entry_add_cascade_ptr (cascade_set%entry(i), cascade, ok)
         end if
      else
         cascade_set%entry(i)%hashval = hashval
         allocate (cascade_set%entry(i)%key (size (key)))
         cascade_set%entry(i)%key = key
         call hash_entry_add_cascade_ptr (cascade_set%entry(i), cascade, ok)
         cascade_set%n_entries = cascade_set%n_entries + 1
      end if
    end subroutine cascade_set_hash_insert_rec
```

12.4.8 External particles

We want to initialize the cascade set with the outgoing particles. In case of multiple processes, initial cascades are prepared for all of them. The hash array check ensures that no particle appears more than once at the same place.

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_add_outgoing (cascade_set, flv)
      type(cascade_set_t), intent(inout), target :: cascade_set
      type(flavor_t), dimension(:,:), intent(in) :: flv
      integer :: pos, prc, n_out, n_prc
      type(cascade_t), pointer :: cascade
      logical :: ok
      n_out = size (flv, dim=1)
```

```
      n_prc = size (flv, dim=2)
      do prc = 1, n_prc
         do pos = 1, n_out
            allocate (cascade)
            call cascade_init_outgoing &
                 (cascade, flv(pos,prc), pos, cascade_set%m_threshold_s)
            call cascade_set_add (cascade_set, cascade, ok)
            if (.not. ok) then
               deallocate (cascade)
            end if
         end do
      end do
   end subroutine cascade_set_add_outgoing
```

The incoming particles are added one at a time. Nevertheless, we may have
several processes which are looped over. At the first opportunity, we set the
pointer `first_t` in the cascade set which should point to the first t-channel
cascade.

Return the indices of the first and last cascade generated.

⟨*Cascades: procedures*⟩+≡

```
   subroutine cascade_set_add_incoming (cascade_set, n1, n2, pos, flv)
     type(cascade_set_t), intent(inout), target :: cascade_set
     integer, intent(out) :: n1, n2
     integer, intent(in) :: pos
     type(flavor_t), dimension(:), intent(in) :: flv
     integer :: prc, n_prc
     type(cascade_t), pointer :: cascade
     logical :: ok
     n1 = 0
     n2 = 0
     n_prc = size (flv)
     do prc = 1, n_prc
        allocate (cascade)
        call cascade_init_incoming &
             (cascade, flv(prc), pos, cascade_set%m_threshold_t)
        call cascade_set_add (cascade_set, cascade, ok)
        if (ok) then
           if (n1 == 0)  n1 = cascade%index
           n2 = cascade%index
           if (.not. associated (cascade_set%first_t)) then
              cascade_set%first_t => cascade
           end if
        else
           deallocate (cascade)
        end if
     end do
   end subroutine cascade_set_add_incoming
```

12.4.9 Cascade combination I: flavor assignment

We have two disjunct cascades, now use the vertex table to determine the pos-
sible flavors of the combination cascade. For each possibility, try to generate a

new cascade. The total cascade depth has to be one less than the limit, because this is reached by setting the keystone.

⟨*Cascades: procedures*⟩+≡
```
subroutine cascade_match_pair (cascade_set, cascade1, cascade2, s_channel)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(cascade_t), intent(in), target :: cascade1, cascade2
  logical, intent(in) :: s_channel
  integer, dimension(:), allocatable :: pdg3
  integer :: i, depth_max
  type(flavor_t) :: flv
  if (s_channel) then
     depth_max = cascade_set%depth_out
  else
     depth_max = cascade_set%depth_tot
  end if
  if (cascade1%depth + cascade2%depth < depth_max) then
     call model_match_vertex (cascade_set%model, &
          flavor_get_pdg (cascade1%flv), &
          flavor_get_pdg (cascade2%flv), &
          pdg3)
     do i = 1, size (pdg3)
        call flavor_init (flv, pdg3(i), cascade_set%model)
        if (s_channel) then
           call cascade_combine_s (cascade_set, cascade1, cascade2, flv)
        else
           call cascade_combine_t (cascade_set, cascade1, cascade2, flv)
        end if
     end do
     deallocate (pdg3)
  end if
end subroutine cascade_match_pair
```

The triplet version takes a third cascade, and we check whether this triplet has a matching vertex in the database. If yes, we make a keystone cascade.

⟨*Cascades: procedures*⟩+≡
```
subroutine cascade_match_triplet &
     (cascade_set, cascade1, cascade2, cascade3, s_channel)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(cascade_t), intent(in), target :: cascade1, cascade2, cascade3
  logical, intent(in) :: s_channel
  integer :: depth_max
  depth_max = cascade_set%depth_tot
  if (cascade1%depth + cascade2%depth + cascade3%depth == depth_max) then
     if (model_check_vertex (cascade_set%model, &
          flavor_get_pdg (cascade1%flv), &
          flavor_get_pdg (cascade2%flv), &
          flavor_get_pdg (cascade3%flv))) then
        call cascade_combine_keystone &
             (cascade_set, cascade1, cascade2, cascade3, s_channel)
     end if
  end if
end subroutine cascade_match_triplet
```

12.4.10 Cascade combination II: kinematics setup and check

Having three matching flavors, we start constructing the combination cascade.
We look at the mass hierarchies and determine whether the cascade is to be
kept. In passing we set mapping modes, resonance properties and such.

   If successful, the cascade is finalized. For a resonant cascade, we prepare in
addition a copy without the resonance.

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_combine_s (cascade_set, cascade1, cascade2, flv)
      type(cascade_set_t), intent(inout), target :: cascade_set
      type(cascade_t), intent(in), target :: cascade1, cascade2
      type(flavor_t), intent(in) :: flv
      type(cascade_t), pointer :: cascade3, cascade4
      logical :: keep
      keep = .false.
      allocate (cascade3)
      call cascade_init (cascade3, cascade1%depth + cascade2%depth + 1)
      cascade3%bincode = ior (cascade1%bincode, cascade2%bincode)
      cascade3%flv = flavor_anti (flv)
      cascade3%pdg = flavor_get_pdg (cascade3%flv)
      cascade3%is_vector = flavor_get_spin_type (flv) == VECTOR
      cascade3%m_min = cascade1%m_min + cascade2%m_min
      cascade3%m_rea = flavor_get_mass (flv)
      if (cascade3%m_rea > cascade_set%m_threshold_s) then
         cascade3%m_eff = cascade3%m_rea
      end if
      ! Potentially resonant cases [sqrts = m_rea for on-shell decay]
      if (cascade3%m_rea > cascade3%m_min &
           .and. cascade3%m_rea <= cascade_set%sqrts) then
         if (flavor_get_width (flv) /= 0) then
            if (cascade1%on_shell .or. cascade2%on_shell) then
               keep = .true.
               cascade3%mapping = S_CHANNEL
               cascade3%resonant = .true.
            end if
         else
            call warn_decay (flv)
         end if
      ! Collinear and IR singular cases
      else if (cascade3%m_rea < cascade_set%sqrts) then
         ! Massless splitting
         if (cascade1%m_eff == 0 .and. cascade2%m_eff == 0 &
              .and. cascade3%depth <= 3) then
            keep = .true.
            cascade3%log_enhanced = .true.
            if (cascade3%is_vector) then
               if (cascade1%is_vector .and. cascade2%is_vector) then
                  cascade3%mapping = COLLINEAR   ! three-vector-vertex
               else
                  cascade3%mapping = INFRARED    ! vector splitting into matter
               end if
            else
               if (cascade1%is_vector .or. cascade2%is_vector) then
                  cascade3%mapping = COLLINEAR   ! vector radiation off matter
```

```
               else
                  cascade3%mapping = INFRARED    ! scalar radiation/splitting
               end if
            end if
      ! IR radiation off massive particle
      else if (cascade3%m_eff > 0 .and. cascade1%m_eff > 0 &
            .and. cascade2%m_eff == 0 &
            .and. (cascade1%on_shell .or. cascade1%mapping == RADIATION) &
            .and. abs (cascade3%m_eff - cascade1%m_eff) &
                     < cascade_set%m_threshold_s) &
            then
         keep = .true.
         cascade3%log_enhanced = .true.
         cascade3%mapping = RADIATION
      else if (cascade3%m_eff > 0 .and. cascade2%m_eff > 0 &
            .and. cascade1%m_eff == 0 &
            .and. (cascade2%on_shell .or. cascade2%mapping == RADIATION) &
            .and. abs (cascade3%m_eff - cascade2%m_eff) &
                  < cascade_set%m_threshold_s) &
            then
         keep = .true.
         cascade3%log_enhanced = .true.
         cascade3%mapping = RADIATION
      end if
   end if
   ! Non-singular cases, including failed resonances
   if (.not. keep) then
      ! Two on-shell particles from a virtual mother
      if (cascade1%on_shell .or. cascade2%on_shell) then
         keep = .true.
         cascade3%m_eff = max (cascade3%m_min, &
                                cascade1%m_eff + cascade2%m_eff)
         if (cascade3%m_eff < cascade_set%m_threshold_s) then
            cascade3%m_eff = 0
         end if
      end if
   end if
   ! Complete and register the cascade (two in case of resonance)
   if (keep) then
      cascade3%on_shell = cascade3%resonant .or. cascade3%log_enhanced
      if (cascade3%resonant) then
         cascade3%pdg = flavor_get_pdg (cascade3%flv)
         allocate (cascade4)
         cascade4 = cascade3
         cascade4%index = cascade_index ()
         call cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
         cascade4%pdg = UNDEFINED
         cascade4%mapping = NO_MAPPING
         cascade4%resonant = .false.
         cascade4%on_shell = .false.
         call cascade_fusion (cascade_set, cascade1, cascade2, cascade4)
      else
         call cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
      end if
```

```
      else
         deallocate (cascade3)
      end if
   contains
     subroutine warn_decay (flv)
       type(flavor_t), intent(in) :: flv
       integer :: i
       integer, dimension(MAX_WARN_RESONANCE), save :: warned_code = 0
       LOOP_WARNED: do i = 1, MAX_WARN_RESONANCE
          if (warned_code(i) == 0) then
             warned_code(i) = flavor_get_pdg (flv)
             write (msg_buffer, "(A)") &
                  & " Intermediate decay of zero-width particle " &
                  & // char (flavor_get_name (flv)) &
                  & // " may be possible."
             call msg_warning
             exit LOOP_WARNED
          else if (warned_code(i) == flavor_get_pdg (flv)) then
             exit LOOP_WARNED
          end if
       end do LOOP_WARNED
     end subroutine warn_decay
   end subroutine cascade_combine_s
```

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: MAX_WARN_RESONANCE = 50
```

This is the t-channel version. `cascade1` is t-channel and contains the seed, `cascade2` is s-channel. We check for kinematically allowed beam decay (which is a fatal error), or massless splitting / soft radiation. The cascade is kept in all remaining cases and submitted for registration.

⟨*Cascades: procedures*⟩+≡
```
   subroutine cascade_combine_t (cascade_set, cascade1, cascade2, flv)
     type(cascade_set_t), intent(inout), target :: cascade_set
     type(cascade_t), intent(in), target :: cascade1, cascade2
     type(flavor_t), intent(in) :: flv
     type(cascade_t), pointer :: cascade3
     allocate (cascade3)
     call cascade_init (cascade3, cascade1%depth + cascade2%depth + 1)
     cascade3%bincode = ior (cascade1%bincode, cascade2%bincode)
     cascade3%flv = flavor_anti (flv)
     cascade3%pdg = flavor_get_pdg (cascade3%flv)
     cascade3%is_vector = flavor_get_spin_type (flv) == VECTOR
     if (cascade1%incoming) then
        cascade3%m_min = cascade2%m_min
     else
        cascade3%m_min = cascade1%m_min + cascade2%m_min
     end if
     cascade3%m_rea = flavor_get_mass (flv)
     if (cascade3%m_rea > cascade_set%m_threshold_t) then
        cascade3%m_eff = max (cascade3%m_rea, cascade2%m_eff)
     else if (cascade2%m_eff > cascade_set%m_threshold_t) then
        cascade3%m_eff = cascade2%m_eff
     else
```

919

```
            cascade3%m_eff = 0
      end if
      ! Allowed decay of beam particle
      if (cascade1%incoming &
            .and. cascade1%m_rea > cascade2%m_rea + cascade3%m_rea) then
            call beam_decay (cascade_set%fatal_beam_decay)
      ! Massless splitting
      else if (cascade1%m_eff == 0 &
            .and. cascade2%m_eff < cascade_set%m_threshold_t &
            .and. cascade3%m_eff == 0) then
         cascade3%mapping = U_CHANNEL
         cascade3%log_enhanced = .true.
      ! IR radiation off massive particle
      else if (cascade1%m_eff /= 0 .and. cascade2%m_eff == 0 &
            .and. cascade3%m_eff /= 0 &
            .and. (cascade1%on_shell .or. cascade1%mapping == RADIATION) &
            .and. abs (cascade1%m_eff - cascade3%m_eff) &
                  < cascade_set%m_threshold_t) &
            then
         cascade3%pdg = flavor_get_pdg (flv)
         cascade3%log_enhanced = .true.
         cascade3%mapping = RADIATION
      end if
      cascade3%t_channel = .true.
      call cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
   contains
      subroutine beam_decay (fatal_beam_decay)
        logical, intent(in) :: fatal_beam_decay
        write (msg_buffer, "(1x,A,1x,'->',1x,A,1x,A)") &
              char (flavor_get_name (cascade1%flv)), &
              char (flavor_get_name (cascade3%flv)), &
              char (flavor_get_name (cascade2%flv))
        call msg_message
        write (msg_buffer, "(1x,'mass(',A,') =',1x,E17.10)") &
              char (flavor_get_name (cascade1%flv)), cascade1%m_rea
        call msg_message
        write (msg_buffer, "(1x,'mass(',A,') =',1x,E17.10)") &
              char (flavor_get_name (cascade3%flv)), cascade3%m_rea
        call msg_message
        write (msg_buffer, "(1x,'mass(',A,') =',1x,E17.10)") &
              char (flavor_get_name (cascade2%flv)), cascade2%m_rea
        call msg_message
        if (fatal_beam_decay) then
           call msg_fatal (" Phase space: Initial beam particle can decay")
        else
           call msg_warning (" Phase space: Initial beam particle can decay")
        end if
      end subroutine beam_decay
   end subroutine cascade_combine_t
```

Here we complete a decay cascade. The third input is the single-particle cascade for the initial particle. There is no resonance or mapping assignment. The only condition for keeping the cascade is the mass sum of the final state, which must

be less than the available energy.

Two modifications are necessary for scattering cascades: a pure s-channel diagram (cascade1 is the incoming particle) do not have a logarithmic mapping at top-level. And in a t-channel diagram, the last line exchanged is mapped t-channel, not u-channel. In both cases we register a new cascade with the modified mapping.

⟨*Cascades: procedures*⟩+≡

```
    subroutine cascade_combine_keystone &
        (cascade_set, cascade1, cascade2, cascade3, s_channel)
      type(cascade_set_t), intent(inout), target :: cascade_set
      type(cascade_t), intent(in), target :: cascade1, cascade2, cascade3
      logical, intent(in) :: s_channel
      type(cascade_t), pointer :: cascade4, cascade0
      logical :: keep, ok
      keep = .false.
      allocate (cascade4)
      call cascade_init &
          (cascade4, cascade1%depth + cascade2%depth + cascade3%depth)
      cascade4%complete = .true.
!     cascade4%bincode = ior (ior (cascade1%bincode, cascade2%bincode), &
!         cascade3%bincode)
      if (s_channel) then
         cascade4%bincode = ior (cascade1%bincode, cascade2%bincode)
      else
         cascade4%bincode = cascade3%bincode
      end if
      cascade4%flv = cascade3%flv
      cascade4%pdg = cascade3%pdg
      cascade4%is_vector = cascade3%is_vector
      cascade4%m_min = cascade1%m_min + cascade2%m_min
      cascade4%m_rea = cascade3%m_rea
      cascade4%m_eff = cascade3%m_rea
      if (cascade4%m_min < cascade_set%sqrts) then
         keep = .true.
      end if
      if (keep) then
         if (cascade1%incoming .and. cascade2%log_enhanced) then
            allocate (cascade0)
            cascade0 = cascade2
            cascade0%next => null ()
            cascade0%index = cascade_index ()
            cascade0%mapping = NO_MAPPING
            cascade0%log_enhanced = .false.
            cascade0%n_log_enhanced = cascade0%n_log_enhanced - 1
            cascade0%tree_mapping(cascade0%depth) = NO_MAPPING
            call cascade_keystone &
                (cascade_set, cascade1, cascade0, cascade3, cascade4, ok)
            if (ok)  call cascade_set_add (cascade_set, cascade0, ok)
         else if (cascade1%t_channel .and. cascade1%mapping == U_CHANNEL) then
            allocate (cascade0)
            cascade0 = cascade1
            cascade0%next => null ()
            cascade0%index = cascade_index ()
```

```
            cascade0%mapping = T_CHANNEL
            cascade0%tree_mapping(cascade0%depth) = T_CHANNEL
            call cascade_keystone &
                 (cascade_set, cascade0, cascade2, cascade3, cascade4, ok)
            if (ok)  call cascade_set_add (cascade_set, cascade0, ok)
        else
            call cascade_keystone &
                 (cascade_set, cascade1, cascade2, cascade3, cascade4, ok)
        end if
    else
        deallocate (cascade4)
    end if
end subroutine cascade_combine_keystone
```

12.4.11   Cascade combination III: node connections and tree fusion

Here we assign global tree properties. If the allowed number of off-shell lines is exceeded, discard the new cascade. Otherwise, assign the trees, sort them, and assign connections. Finally, append the cascade to the list. This may fail (because in the hash array there is already an equivalent cascade). On failure, discard the cascade.

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
      type(cascade_set_t), intent(inout), target :: cascade_set
      type(cascade_t), intent(in), target :: cascade1, cascade2
      type(cascade_t), pointer :: cascade3
      integer :: i1, i2, i3, i4
      logical :: ok
      cascade3%internal = (cascade3%depth - 3) / 2
      if (cascade3%resonant) then
         cascade3%multiplicity = 1
         cascade3%n_resonances = &
              cascade1%n_resonances + cascade2%n_resonances + 1
      else
         cascade3%multiplicity = cascade1%multiplicity + cascade2%multiplicity
         cascade3%n_resonances = cascade1%n_resonances + cascade2%n_resonances
      end if
      if (cascade3%log_enhanced) then
         cascade3%n_log_enhanced = &
              cascade1%n_log_enhanced + cascade2%n_log_enhanced + 1
      else
         cascade3%n_log_enhanced = &
              cascade1%n_log_enhanced + cascade2%n_log_enhanced
      end if
      if (cascade3%t_channel) then
         cascade3%n_t_channel = cascade1%n_t_channel + 1
      end if
      if (cascade3%internal - cascade3%n_resonances - cascade3%n_log_enhanced &
           > cascade_set%off_shell) then
         deallocate (cascade3)
      else if (cascade3%n_t_channel > cascade_set%t_channel) then
```

922

```
          deallocate (cascade3)
      else
          i1 = cascade1%depth
          i2 = i1 + 1
          i3 = i1 + cascade2%depth
          i4 = cascade3%depth
          cascade3%tree(:i1) = cascade1%tree
          where (cascade1%tree_mapping /= NO_MAPPING)
             cascade3%tree_pdg(:i1) = cascade1%tree_pdg
          elsewhere
             cascade3%tree_pdg(:i1) = UNDEFINED
          end where
          cascade3%tree_mapping(:i1) = cascade1%tree_mapping
          cascade3%tree_resonant(:i1) = cascade1%tree_resonant
          cascade3%tree(i2:i3) = cascade2%tree
          where (cascade2%tree_mapping /= NO_MAPPING)
             cascade3%tree_pdg(i2:i3) = cascade2%tree_pdg
          elsewhere
             cascade3%tree_pdg(i2:i3) = UNDEFINED
          end where
          cascade3%tree_mapping(i2:i3) = cascade2%tree_mapping
          cascade3%tree_resonant(i2:i3) = cascade2%tree_resonant
          cascade3%tree(i4) = cascade3%bincode
          cascade3%tree_pdg(i4) = cascade3%pdg
          cascade3%tree_mapping(i4) = cascade3%mapping
          cascade3%tree_resonant(i4) = cascade3%resonant
          call tree_sort (cascade3%tree, &
               cascade3%tree_pdg, cascade3%tree_mapping, cascade3%tree_resonant)
          cascade3%has_children = .true.
          cascade3%daughter1 => cascade1
          cascade3%daughter2 => cascade2
          call cascade_set_add (cascade_set, cascade3, ok)
          if (.not. ok)  deallocate (cascade3)
      end if
  end subroutine cascade_fusion
```

Here we combine a cascade pair with an incoming particle, i.e., we set a keystone. Otherwise, this is similar. On the first opportunity, we set the `first_k` pointer in the cascade set.

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_keystone &
      (cascade_set, cascade1, cascade2, cascade3, cascade4, ok)
    type(cascade_set_t), intent(inout), target :: cascade_set
    type(cascade_t), intent(in), target :: cascade1, cascade2, cascade3
    type(cascade_t), pointer :: cascade4
    logical, intent(out) :: ok
    integer :: i1, i2, i3, i4
    cascade4%internal = (cascade4%depth - 3) / 2
    cascade4%multiplicity = cascade1%multiplicity + cascade2%multiplicity
    cascade4%n_resonances = cascade1%n_resonances + cascade2%n_resonances
    cascade4%n_log_enhanced = &
            cascade1%n_log_enhanced + cascade2%n_log_enhanced
    cascade4%n_t_channel = cascade1%n_t_channel + cascade2%n_t_channel + 1
```

```
        if (cascade4%internal - cascade4%n_resonances - cascade4%n_log_enhanced &
             > cascade_set%off_shell) then
           deallocate (cascade4)
           ok = .false.
        else if (cascade4%n_t_channel > cascade_set%t_channel) then
           deallocate (cascade4)
           ok = .false.
        else
           i1 = cascade1%depth
           i2 = i1 + 1
           i3 = i1 + cascade2%depth
           i4 = cascade4%depth
           cascade4%tree(:i1) = cascade1%tree
           where (cascade1%tree_mapping /= NO_MAPPING)
              cascade4%tree_pdg(:i1) = cascade1%tree_pdg
           elsewhere
              cascade4%tree_pdg(:i1) = UNDEFINED
           end where
           cascade4%tree_mapping(:i1) = cascade1%tree_mapping
           cascade4%tree_resonant(:i1) = cascade1%tree_resonant
           cascade4%tree(i2:i3) = cascade2%tree
           where (cascade2%tree_mapping /= NO_MAPPING)
              cascade4%tree_pdg(i2:i3) = cascade2%tree_pdg
           elsewhere
              cascade4%tree_pdg(i2:i3) = UNDEFINED
           end where
           cascade4%tree_mapping(i2:i3) = cascade2%tree_mapping
           cascade4%tree_resonant(i2:i3) = cascade2%tree_resonant
!           cascade4%tree(i4) = cascade3%bincode
           cascade4%tree(i4) = cascade4%bincode
           cascade4%tree_pdg(i4) = UNDEFINED
           cascade4%tree_mapping(i4) = NO_MAPPING
           cascade4%tree_resonant(i4) = .false.
           call tree_sort (cascade4%tree, &
                cascade4%tree_pdg, cascade4%tree_mapping, cascade4%tree_resonant)
           cascade4%has_children = .true.
           cascade4%daughter1 => cascade1
           cascade4%daughter2 => cascade2
           cascade4%mother => cascade3
           call cascade_set_add (cascade_set, cascade4, ok)
           if (ok) then
              if (.not. associated (cascade_set%first_k)) then
                 cascade_set%first_k => cascade4
              end if
           else
              deallocate (cascade4)
           end if
        end if
   end subroutine cascade_keystone
```

Sort a tree (array of binary codes) and particle code array simultaneously, by
ascending binary codes. A convenient method is to use the `maxloc` function
iteratively, to find and remove the largest entry in the tree array one by one.

⟨*Cascades: procedures*⟩+≡
```
    subroutine tree_sort (tree, pdg, mapping, resonant)
      integer(TC), dimension(:), intent(inout) :: tree
      integer, dimension(:), intent(inout) :: pdg, mapping
      logical, dimension(:), intent(inout) :: resonant
      integer(TC), dimension(size(tree)) :: tree_tmp
      integer, dimension(size(pdg)) :: pdg_tmp, mapping_tmp
      logical, dimension(size(resonant)) :: resonant_tmp
      integer, dimension(1) :: pos
      integer :: i
      tree_tmp = tree
      pdg_tmp = pdg
      mapping_tmp = mapping
      resonant_tmp = resonant
      do i = size(tree),1,-1
         pos = maxloc (tree_tmp)
         tree(i) = tree_tmp (pos(1))
         pdg(i) = pdg_tmp (pos(1))
         mapping(i) = mapping_tmp (pos(1))
         resonant(i) = resonant_tmp (pos(1))
         tree_tmp(pos(1)) = 0
      end do
    end subroutine tree_sort
```

12.4.12 Cascade set generation

We use a nested scan to combine all cascades with all other cascades.

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_generate_s (cascade_set)
      type(cascade_set_t), intent(inout), target :: cascade_set
      type(cascade_t), pointer :: cascade1, cascade2
      cascade1 => cascade_set%first
      LOOP1: do while (associated (cascade1))
         cascade2 => cascade_set%first
         LOOP2: do while (associated (cascade2))
            if (cascade2%index >= cascade1%index)  exit LOOP2
            if (cascade1 .disjunct. cascade2) then
               call cascade_match_pair (cascade_set, cascade1, cascade2, .true.)
            end if
            cascade2 => cascade2%next
         end do LOOP2
         cascade1 => cascade1%next
      end do LOOP1
    end subroutine cascade_set_generate_s
```

The t-channel cascades are directed and have a seed (one of the incoming particles) and a target (the other one). We loop over all possible seeds and targets. Inside this, we loop over all t-channel cascades (`cascade1`) and s-channel cascades (`cascade2`) and try to combine them.

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_generate_t (cascade_set, pos_seed, pos_target)
      type(cascade_set_t), intent(inout), target :: cascade_set
```

```
            integer, intent(in) :: pos_seed, pos_target
            type(cascade_t), pointer :: cascade_seed, cascade_target
            type(cascade_t), pointer :: cascade1, cascade2
            integer(TC) :: bc_seed, bc_target
            bc_seed = ibset (0_TC, pos_seed-1)
            bc_target = ibset (0_TC, pos_target-1)
            cascade_seed => cascade_set%first_t
            LOOP_SEED: do while (associated (cascade_seed))
               if (cascade_seed%bincode == bc_seed) then
                  cascade_target => cascade_set%first_t
                  LOOP_TARGET: do while (associated (cascade_target))
                     if (cascade_target%bincode == bc_target) then
                        cascade1 => cascade_set%first_t
                        LOOP_T: do while (associated (cascade1))
                           if ((cascade1 .disjunct. cascade_target) &
                                .and. .not. (cascade1 .disjunct. cascade_seed)) then
                              cascade2 => cascade_set%first
                              LOOP_S: do while (associated (cascade2))
                                 if ((cascade2 .disjunct. cascade_target) &
                                      .and. (cascade2 .disjunct. cascade1)) then
                                    call cascade_match_pair &
                                         (cascade_set, cascade1, cascade2, .false.)
                                 end if
                                 cascade2 => cascade2%next
                              end do LOOP_S
                           end if
                           cascade1 => cascade1%next
                        end do LOOP_T
                     end if
                     cascade_target => cascade_target%next
                  end do LOOP_TARGET
               end if
               cascade_seed => cascade_seed%next
            end do LOOP_SEED
         end subroutine cascade_set_generate_t
```

This part completes the phase space for decay processes. It is similar to s-channel cascade generation, but combines two cascade with the particular cascade of the incoming particle. This particular cascade is expected to be pointed at by `first_t`.

⟨*Cascades: procedures*⟩+≡

```
         subroutine cascade_set_generate_decay (cascade_set)
           type(cascade_set_t), intent(inout), target :: cascade_set
           type(cascade_t), pointer :: cascade1, cascade2
           type(cascade_t), pointer :: cascade_in
           cascade_in => cascade_set%first_t
           cascade1 => cascade_set%first
           do while (associated (cascade1))
              if (cascade1 .disjunct. cascade_in) then
                 cascade2 => cascade1%next
                 do while (associated (cascade2))
                    if ((cascade2 .disjunct. cascade1) &
                         .and. (cascade2 .disjunct. cascade_in)) then
```

```
                        call cascade_match_triplet (cascade_set, &
                            cascade1, cascade2, cascade_in, .true.)
                  end if
                  cascade2 => cascade2%next
               end do
            end if
            cascade1 => cascade1%next
         end do
   end subroutine cascade_set_generate_decay
```

This part completes the phase space for scattering processes. We combine a t-channel cascade (containing the seed) with a s-channel cascade and the target.

⟨*Cascades: procedures*⟩+≡

```
   subroutine cascade_set_generate_scattering &
        (cascade_set, ns1, ns2, nt1, nt2, pos_seed, pos_target)
     type(cascade_set_t), intent(inout), target :: cascade_set
     integer, intent(in) :: pos_seed, pos_target
     integer, intent(in) :: ns1, ns2, nt1, nt2
     type(cascade_t), pointer :: cascade_seed, cascade_target
     type(cascade_t), pointer :: cascade1, cascade2
     integer(TC) :: bc_seed, bc_target
     bc_seed = ibset (0_TC, pos_seed-1)
     bc_target = ibset (0_TC, pos_target-1)
     cascade_seed => cascade_set%first_t
     LOOP_SEED: do while (associated (cascade_seed))
        if (cascade_seed%index < ns1) then
           cascade_seed => cascade_seed%next
           cycle LOOP_SEED
        else if (cascade_seed%index > ns2) then
           exit LOOP_SEED
        else if (cascade_seed%bincode == bc_seed) then
           cascade_target => cascade_set%first_t
           LOOP_TARGET: do while (associated (cascade_target))
              if (cascade_target%index < nt1) then
                 cascade_target => cascade_target%next
                 cycle LOOP_TARGET
              else if (cascade_target%index > nt2) then
                 exit LOOP_TARGET
              else if (cascade_target%bincode == bc_target) then
                 cascade1 => cascade_set%first_t
                 LOOP_T: do while (associated (cascade1))
                    if ((cascade1 .disjunct. cascade_target) &
                         .and. .not. (cascade1 .disjunct. cascade_seed)) then
                       cascade2 => cascade_set%first
                       LOOP_S: do while (associated (cascade2))
                          if ((cascade2 .disjunct. cascade_target) &
                               .and. (cascade2 .disjunct. cascade1)) then
                             call cascade_match_triplet (cascade_set, &
                                  cascade1, cascade2, cascade_target, .false.)
                          end if
                          cascade2 => cascade2%next
                       end do LOOP_S
                    end if
```

```
              cascade1 => cascade1%next
            end do LOOP_T
          end if
          cascade_target => cascade_target%next
        end do LOOP_TARGET
      end if
      cascade_seed => cascade_seed%next
    end do LOOP_SEED
  end subroutine cascade_set_generate_scattering
```

12.4.13 Groves

After all cascades are recorded, we group the complete cascades in groves. A grove consists of cascades with identical multiplicity, number of resonances, log-enhanced, and t-channel lines.

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_assign_groves (cascade_set)
      type(cascade_set_t), intent(inout), target :: cascade_set
      type(cascade_t), pointer :: cascade1, cascade2
      integer :: multiplicity, n_resonances, n_log_enhanced, n_t_channel
      integer :: grove
      grove = 0
      cascade1 => cascade_set%first_k
      do while (associated (cascade1))
         if (cascade1%active .and. cascade1%complete &
              .and. cascade1%grove == 0) then
            grove = grove + 1
            cascade1%grove = grove
            multiplicity = cascade1%multiplicity
            n_resonances = cascade1%n_resonances
            n_log_enhanced = cascade1%n_log_enhanced
            n_t_channel = cascade1%n_t_channel
            cascade2 => cascade1%next
            do while (associated (cascade2))
               if (cascade2%grove == 0) then
                  if (cascade2%multiplicity == multiplicity &
                       .and. cascade2%n_resonances == n_resonances &
                       .and. cascade2%n_log_enhanced == n_log_enhanced &
                       .and. cascade2%n_t_channel == n_t_channel) then
                     cascade2%grove = grove
                  end if
               end if
               cascade2 => cascade2%next
            end do
         end if
         cascade1 => cascade1%next
      end do
      cascade_set%n_groves = grove
    end subroutine cascade_set_assign_groves
```

12.4.14 Generate the phase space file

Generate a complete phase space configuration. First, all s-channel graphs that can be built up from the outgoing particles. Then we distinguish (1) decay, where we complete the s-channel graphs by connecting to the input line, and (2) scattering, where we now generate t-channel graphs by introducing an incoming particle, and complete this by connecting to the other incoming particle.

⟨*Cascades: public*⟩+≡
```
    public :: cascade_set_generate
```

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_generate &
         (cascade_set, model, n_in, n_out, flv, phs_par, fatal_beam_decay)
      type(cascade_set_t), intent(out) :: cascade_set
      type(model_t), intent(in), target :: model
      integer, intent(in) :: n_in, n_out
      type(flavor_t), dimension(:,:), intent(in) :: flv
      type(phs_parameters_t), intent(in) :: phs_par
      integer :: n11, n12, n21, n22
      logical, intent(in) :: fatal_beam_decay
      if (phase_space_vanishes (phs_par%sqrts, n_in, flv))  return
      call cascade_set_init (cascade_set, model, n_in, n_out, phs_par, &
         fatal_beam_decay)
      call cascade_set_add_outgoing (cascade_set, flv(n_in+1:,:))
      call cascade_set_generate_s (cascade_set)
      select case (n_in)
      case(1)
         call cascade_set_add_incoming &
              (cascade_set, n11, n12, n_out + 1, flv(1,:))
         call cascade_set_generate_decay (cascade_set)
      case(2)
         call cascade_set_add_incoming &
              (cascade_set, n11, n12, n_out + 1, flv(2,:))
         call cascade_set_add_incoming &
              (cascade_set, n21, n22, n_out + 2, flv(1,:))
         call cascade_set_generate_t (cascade_set, n_out + 1, n_out + 2)
         call cascade_set_generate_t (cascade_set, n_out + 2, n_out + 1)
         call cascade_set_generate_scattering &
              (cascade_set, n11, n12, n21, n22, n_out + 1, n_out + 2)
         call cascade_set_generate_scattering &
              (cascade_set, n21, n22, n11, n12, n_out + 2, n_out + 1)
      end select
      call cascade_set_assign_groves (cascade_set)
    end subroutine cascade_set_generate
```

Sanity check: Before anything else is done, check if there could possibly be any phase space.

⟨*Cascades: procedures*⟩+≡
```
    function phase_space_vanishes (sqrts, n_in, flv) result (flag)
      logical :: flag
      real(default), intent(in) :: sqrts
      integer, intent(in) :: n_in
      type(flavor_t), dimension(:,:), intent(in) :: flv
      real(default), dimension(:,:), allocatable :: mass
```

```
      real(default), dimension(:), allocatable :: mass_in, mass_out
      integer :: n_prt, n_flv
      flag = .false.
      if (sqrts <= 0) then
         call msg_error ("Phase space vanishes (sqrts must be positive)")
         flag = .true.;   return
      end if
      n_prt = size (flv, 1)
      n_flv = size (flv, 2)
      allocate (mass (n_prt, n_flv), mass_in (n_flv), mass_out (n_flv))
      mass = flavor_get_mass (flv)
      mass_in = sum (mass(:n_in,:), 1)
      mass_out = sum (mass(n_in+1:,:), 1)
      if (any (mass_in > sqrts)) then
         call msg_error ("Mass sum of incoming particles " &
              // "is more than available energy")
         flag = .true.;   return
      end if
      if (any (mass_out > sqrts)) then
         call msg_error ("Mass sum of outgoing particles " &
              // "is more than available energy")
         flag = .true.;   return
      end if
    end function phase_space_vanishes
```

12.4.15   Test

⟨*Cascades: public*⟩+≡
```
    public :: cascade_test
```

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_test
      use os_interface, only: os_data_t
      type(os_data_t) :: os_data
      type(model_t), pointer :: model
      type(flavor_t), dimension(5,2) :: flv
      type(cascade_set_t) :: cascade_set
      type(string_t) :: name, filename
      type(phs_parameters_t) :: phs_par
      name = "QCD"
      filename = "test.mdl"
      call syntax_model_file_init ()
      call model_list_read_model (name, filename, os_data, model)
      call model_write (model, verbose=.true.)
      call flavor_init (flv(1,1), 2, model)
      call flavor_init (flv(2,1),-2, model)
      call flavor_init (flv(3,1), 1, model)
      call flavor_init (flv(4,1),-1, model)
      call flavor_init (flv(5,1),21, model)
      call flavor_init (flv(1,2), 2, model)
      call flavor_init (flv(2,2),-2, model)
      call flavor_init (flv(3,2), 2, model)
      call flavor_init (flv(4,2),-2, model)
```

```
      call flavor_init (flv(5,2),21, model)
      phs_par%sqrts = 1000._default
      phs_par%off_shell = 2
      call cascade_set_generate (cascade_set, model, 2, 3, flv, phs_par,.true.)
      call cascade_set_write (cascade_set)
      call cascade_set_write_file_format (cascade_set)
      call cascade_set_final (cascade_set)
      call model_list_final ()
    end subroutine cascade_test
```

Chapter 13

Integration and event generation

With all necessary ingredients set up in the previous modules, the modules in this chapter do the high-level work of interfacing the process library, integrating and generating events.

process_libraries Create a process library, compile it, and load it, providing procedure pointers that the following modules can use.

hard_interactions Set up hard matrix element data appropriate for a given process and evaluate matrix elements in various ways (traced over all quantum numbers for integration, exclusive in some quantum numbers for analysis, expanded in color-flow patterns for showering).

processes Collect phase space, hard interactions, structure functions, cuts and whatever is needed to compute total partonic cross sections and partial decay widths.

events In the simulation step, generate individual events, concatenate decays, etc.

13.0.16   Process library interface

The process library is generated dynamically, and it does not provide a module. To access it, we define explicit interfaces that have to be included in the calling modules. These interfaces must be accessed both at compile time and at runtime, therefore we define a separate module that can be installed independent of the rest.

⟨`prclib_interfaces.f90`⟩≡
  ⟨*File header*⟩

  module prclib_interfaces

  ⟨*Use strings*⟩
    use iso_c_binding !NODEP!
    use kinds !NODEP!

⟨*Standard module head*⟩

⟨*Prclib interfaces: public*⟩

⟨*Prclib interfaces: interfaces*⟩

```
end module prclib_interfaces
```

Return the number of processes contained in the library.

⟨*Prclib interfaces: public*⟩≡
```
    public :: prc_get_n_processes
```

⟨*Prclib interfaces: interfaces*⟩≡
```
    abstract interface
       function prc_get_n_processes () result (n) bind(C)
         import
         integer(c_int) :: n
       end function prc_get_n_processes
    end interface
```

Return the C pointer to a string, and its length.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_get_stringptr
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
       subroutine prc_get_stringptr (i, cptr, len) bind(C)
         import
         integer(c_int), intent(in) :: i
         type(c_ptr), intent(out) :: cptr
         integer(c_int), intent(out) :: len
       end subroutine prc_get_stringptr
    end interface
```

Return an integer.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_get_int
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
       function prc_get_int (pid) result (n) bind(C)
         import
         integer(c_int), intent(in) :: pid
         integer(c_int) :: n
       end function prc_get_int
    end interface
```

Return a two-dimensional integer array

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_set_int_tab1
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
       subroutine prc_set_int_tab1 (pid, cptr, shape) bind(C)
         import
         integer(c_int), intent(in) :: pid
```

```
            type(c_ptr), intent(in) :: cptr
            integer(c_int), dimension(2), intent(in) :: shape
          end subroutine prc_set_int_tab1
        end interface
```

Return a three-dimensional integer and a two-dimensional logical array.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_set_int_tab2
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
      subroutine prc_set_int_tab2 (pid, cptr, shape, lcptr, lshape) bind(C)
        import
        integer(c_int), intent(in) :: pid
        type(c_ptr), intent(in) :: cptr
        integer(c_int), dimension(3), intent(in) :: shape
        type(c_ptr), intent(in) :: lcptr
        integer(c_int), dimension(2), intent(in) :: lshape
      end subroutine prc_set_int_tab2
    end interface
```

Return two integer arrays and a complex array.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_set_cf_tab
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
      subroutine prc_set_cf_tab (pid, iptr1, iptr2, cptr, shape) bind(C)
        import
        integer(c_int), intent(in) :: pid
        type(c_ptr), intent(in) :: iptr1, iptr2, cptr
        integer(c_int), dimension(1), intent(in) :: shape
      end subroutine prc_set_cf_tab
    end interface
```

These procedure signatures correspond to the process-specific API. The actual procedures are assigned by the pointer-assignment functions with signatures below.

Do overall process initialization if necessary.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_init
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
      subroutine prc_init (par) bind(C)
        import
        real(c_default_float), dimension(*), intent(in) :: par
      end subroutine prc_init
    end interface
```

Do overall process finalization if necessary.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_final
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
       subroutine prc_final () bind(C)
       end subroutine prc_final
    end interface
```

Update the α_s value used for the matrix element computation.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_update_alpha_s
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
  interface
       subroutine prc_update_alpha_s (alpha_s) bind(C)
         import
         real(c_default_float), intent(in) :: alpha_s
       end subroutine prc_update_alpha_s
    end interface
```

Reset the counters for individual helicities. momenta.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_reset_helicity_selection
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
       subroutine prc_reset_helicity_selection (threshold, cutoff) bind(C)
         import
         real(c_default_float), intent(in) :: threshold
         integer(c_int), intent(in) :: cutoff
       end subroutine prc_reset_helicity_selection
    end interface
```

Request the calculation of a new event, given a set of particle momenta.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_new_event
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
       subroutine prc_new_event (p) bind(C)
         import
         real(c_default_float), dimension(0:3,*), intent(in) :: p
       end subroutine prc_new_event
    end interface
```

Return true if the selected combination of flavor, helicity and color is allowed.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_is_allowed
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
       function prc_is_allowed (flv, hel, col) result (is_allowed) bind(C)
         import
         logical(c_bool) :: is_allowed
         integer(c_int), intent(in) :: flv, hel, col
       end function prc_is_allowed
    end interface
```

Assuming that the event has been computed, return a particular amplitude.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_get_amplitude
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
       function prc_get_amplitude (flv, hel, col) result (amp) bind(C)
         import
         complex(c_default_complex) :: amp
         integer(c_int), intent(in) :: flv, hel, col
       end function prc_get_amplitude
    end interface
```

Function that returns the pointer to a procedure:

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_get_fptr
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
       subroutine prc_get_fptr (pid, fptr) bind(C)
         import
         integer(c_int), intent(in) :: pid
         type(c_funptr), intent(out) :: fptr
       end subroutine prc_get_fptr
    end interface
```

Interface for additional library unload / reload hooks

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prclib_unload_hook
    public :: prclib_reload_hook
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
       subroutine prclib_unload_hook (libname)
         import
         type(string_t), intent(in) :: libname
       end subroutine prclib_unload_hook

       subroutine prclib_reload_hook (libname)
         import
         type(string_t), intent(in) :: libname
       end subroutine prclib_reload_hook
    end interface
```

13.1   Process library access

This module interfaces the OS to create, build, and use process libraries.

⟨process_libraries.f90⟩≡
```
  ⟨File header⟩

  module process_libraries

    use iso_c_binding !NODEP!
    use kinds !NODEP!
  ⟨Use strings⟩
  ⟨Use file utils⟩
    use diagnostics !NODEP!
```

```
    use md5
    use os_interface
    use lexers
    use variables
    use models
    use flavors
    use prclib_interfaces
```

⟨*Standard module head*⟩

⟨*Process libraries: public*⟩

⟨*Process libraries: parameters*⟩

⟨*Process libraries: types*⟩

⟨*Process libraries: interfaces*⟩

⟨*Process libraries: variables*⟩

```
    contains
```

⟨*Process libraries: procedures*⟩

```
    end module process_libraries
```

13.1.1   Process methods

⟨*Process libraries: parameters*⟩≡
```
    integer, parameter, public :: PRC_UNDEFINED = 0
    integer, parameter, public :: PRC_OMEGA = 1
    integer, parameter, public :: PRC_TEST = 2
    integer, parameter, public :: PRC_UNIT = 3
    integer, parameter, public :: PRC_EXTERNAL = 4
    integer, parameter, public :: PRC_DIPOLE = 5
```

13.1.2   Status codes

⟨*Process libraries: parameters*⟩+≡
```
    integer, parameter :: STAT_UNKNOWN = 0
    integer, parameter :: STAT_CONFIGURED = 1
    integer, parameter :: STAT_CODE_GENERATED = 2
    integer, parameter :: STAT_COMPILED = 3
    integer, parameter :: STAT_LOADED = 4
    integer, parameter :: STAT_INTEGRATED = 5
```

13.1.3   Process configuration data

Process configuration data.

⟨*Process libraries: public*⟩≡
```
    public :: process_configuration_t
```

⟨*Process libraries: types*⟩≡
```
type :: process_configuration_t
  private
  integer :: status = STAT_UNKNOWN
  integer :: method = PRC_UNDEFINED
  type(string_t) :: id
  type(model_t), pointer :: model => null ()
  integer :: n_in  = 0
  integer :: n_out = 0
  integer :: n_tot = 0
  type(string_t), dimension(:), allocatable :: prt_in, prt_out
  type(string_t) :: restrictions
  character(32) :: md5sum = ""
  logical :: result_is_known = .false.
  integer :: n_calls = 0
  real(default) :: integral = 0
  real(default) :: error = 0
  real(default) :: accuracy = 0
  real(default) :: chi2 = 0
  real(default) :: efficiency = 0
  type(process_configuration_t), pointer :: next => null ()
end type process_configuration_t
```

Initialize a process configuration. The configuration is `intent(inout)` such that
any `next` pointer is kept if an existing configuration is overwritten. Otherwise,
all contents are reset.

⟨*Process libraries: procedures*⟩≡
```
subroutine process_configuration_init &
    (prc_conf, prc_id, model, prt_in, prt_out, method, status, &
     restrictions, known_md5sum)
  type(process_configuration_t), intent(inout) :: prc_conf
  type(string_t), intent(in) :: prc_id
  type(model_t), intent(in), target :: model
  type(string_t), dimension(:), intent(in) :: prt_in, prt_out
  integer, intent(in), optional :: status
  integer, intent(in), optional :: method
  type(string_t), intent(in), optional :: restrictions
  character(32), intent(in), optional :: known_md5sum
  prc_conf%id = prc_id
  prc_conf%model => model
  prc_conf%n_in  = size (prt_in)
  prc_conf%n_out = size (prt_out)
  prc_conf%n_tot = prc_conf%n_in + prc_conf%n_out
  if (allocated (prc_conf%prt_in))  deallocate (prc_conf%prt_in)
  allocate (prc_conf%prt_in  (prc_conf%n_in))
  if (allocated (prc_conf%prt_out)) deallocate (prc_conf%prt_out)
  allocate (prc_conf%prt_out (prc_conf%n_out))
  prc_conf%prt_in  = prt_in
  prc_conf%prt_out = prt_out
  prc_conf%result_is_known = .false.
  prc_conf%n_calls = 0
  prc_conf%integral = 0
  prc_conf%error = 0
```

```
      prc_conf%accuracy = 0
      prc_conf%chi2 = 0
      prc_conf%efficiency = 0
      if (present (status)) then
         prc_conf%status = status
      else
         prc_conf%status = STAT_CONFIGURED
      end if
      if (present (method)) then
         prc_conf%method = method
      else
         prc_conf%method = PRC_OMEGA
      end if
      if (present (restrictions)) then
         prc_conf%restrictions = canonicalize_restrictions (restrictions, model)
      else
         prc_conf%restrictions = ""
      end if
      if (present (known_md5sum)) then
         prc_conf%md5sum = known_md5sum
      else
         call process_configuration_compute_md5sum (prc_conf)
      end if
   end subroutine process_configuration_init
```

Compute the MD5 sum. Write all relevant information to a string.

⟨*Process libraries: procedures*⟩+≡
```
   subroutine process_configuration_compute_md5sum (prc_conf)
     type(process_configuration_t), intent(inout) :: prc_conf
     integer :: u, i
     u = free_unit ()
     open (unit=u, status="scratch")
     write (u, "(A)")  char (model_get_name (prc_conf%model))
     write (u, "(I0)")  prc_conf%n_in
     write (u, "(I0)")  prc_conf%n_out
     write (u, "(I0)")  prc_conf%n_tot
     do i = 1, size (prc_conf%prt_in)
        write (u, "(A)")  char (prc_conf%prt_in(i))
     end do
     do i = 1, size (prc_conf%prt_out)
        write (u, "(A)")  char (prc_conf%prt_out(i))
     end do
     if (prc_conf%restrictions /= "") then
        write (u, "(A)")  char (prc_conf%restrictions)
     end if
     rewind (u)
     prc_conf%md5sum = md5sum (u)
     close (u)
   end subroutine process_configuration_compute_md5sum
```

Record the results from an integration run and mark the process as integrated.

⟨*Process libraries: procedures*⟩+≡
```
   subroutine process_configuration_record_integral &
```

```
        (prc_conf, n_calls, integral, error, accuracy, chi2, efficiency)
      type(process_configuration_t), intent(inout) :: prc_conf
      integer, intent(in) :: n_calls
      real(default), intent(in) :: integral, error, accuracy, chi2, efficiency
      prc_conf%n_calls = n_calls
      prc_conf%integral = integral
      prc_conf%error = error
      prc_conf%accuracy = accuracy
      prc_conf%chi2 = chi2
      prc_conf%efficiency = efficiency
      prc_conf%result_is_known = .true.
      prc_conf%status = STAT_INTEGRATED
    end subroutine process_configuration_record_integral
```

Output (used by the 'list' command):

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_configuration_write (prc_conf, unit)
      type(process_configuration_t), intent(in) :: prc_conf
      integer, intent(in), optional :: unit
      character :: status
      type(string_t) :: in_state, out_state
      integer :: i
      select case (prc_conf%status)
      case (STAT_UNKNOWN);         status = "?"
      case (STAT_CONFIGURED);      status = "O"
      case (STAT_CODE_GENERATED);  status = "G"
      case (STAT_COMPILED);        status = "C"
      case (STAT_LOADED);          status = "L"
      case (STAT_INTEGRATED);      status = "I"
      end select
      in_state = prc_conf%prt_in(1)
      do i = 2, size (prc_conf%prt_in)
         in_state = in_state // ", " // prc_conf%prt_in(i)
      end do
      out_state = prc_conf%prt_out(1)
      do i = 2, size (prc_conf%prt_out)
         out_state = out_state // ", " // prc_conf%prt_out(i)
      end do
      if (prc_conf%restrictions == "") then
         call msg_message (" [" // status // "] " // char (prc_conf%id) // " = " &
              // char (in_state) // " => " // char (out_state), unit)
      else
         call msg_message (" [" // status // "] " // char (prc_conf%id) // " = " &
              // char (in_state) // " => " // char (out_state) &
              // " { $restrictions = " // '"' // char (prc_conf%restrictions) &
              // '"' // " }", unit)  ! $
      end if
    end subroutine process_configuration_write
```

13.1.4 Canonicalize particle names

The `restrictions` string can contain particle names, but if it is passed verbatim
to the matrix element generator, only those particles that the latter knows will
be understood. Therefore, we tokenize the string, translate the particle names,
and return the string with translations.

⟨*Process libraries: procedures*⟩+≡

```
    function canonicalize_restrictions (string, model) result (newstring)
      type(string_t) :: newstring
      type(string_t), intent(in) :: string
      type(model_t), intent(in), target :: model
      type(stream_t), target :: stream
      type(lexer_t) :: lexer
      type(lexeme_t) :: lexeme
      type(string_t) :: token
      newstring = ""
      if (string == "")  return
      call lexer_init (lexer, &
           comment_chars = "", &
           quote_chars = "’", &
           quote_match = "’", &
           single_chars = "+~", &
           special_class = (/ "&" /), &
           keyword_list = null ())
      call stream_init (stream, string)
      call lexer_assign_stream (lexer, stream)
      TRANSFORM_TOKENS: do
         call lex (lexeme, lexer)
         if (lexeme_is_eof (lexeme))  exit TRANSFORM_TOKENS
         if (lexeme_is_break (lexeme)) then
            call msg_message ("Restriction string = " &
                 // ’"’ // char (string) // ’"’)
            call msg_fatal ("Syntax error in restrictions specification")
            exit TRANSFORM_TOKENS
         end if
         token = lexeme_get_contents (lexeme)
         select case (lexeme_get_type (lexeme))
         case (T_NUMERIC)
            newstring = newstring // token
         case (T_IDENTIFIER)
            select case (char (extract (token, 1, 1)))
            case ("+", "~", "&")
               newstring = newstring // token
            case default
               newstring = newstring // canonicalize_prt (token, model)
            end select
         case (T_QUOTED)
            newstring = newstring // canonicalize_prt (token, model)
         case default
            call msg_bug ("Token type error in restrictions specification")
         end select
      end do TRANSFORM_TOKENS
      call stream_final (stream)
    end function canonicalize_restrictions
```

Transform a particle string into its flavor code and back; this yields the canonical name.

⟨*Process libraries: procedures*⟩+≡
```
    function canonicalize_prt (string, model) result (newstring)
      type(string_t) :: newstring
      type(string_t), intent(in) :: string
      type(model_t), intent(in), target :: model
      type(flavor_t) :: flv
      integer :: pdg
      pdg = model_get_particle_pdg (model, string)
      if (pdg == 0) then
         call msg_fatal ("Undefined particle in restrictions specification")
      end if
      call flavor_init (flv, pdg, model)
      newstring = flavor_get_name (flv)
    end function canonicalize_prt
```

13.1.5   Process library data

This object contains filenames, the complete set of process configuration data, the C filehandle interface for the shared library, and procedure pointers for the library functions.

The contents of this type are public because we do not want to have another wrapper around the procedure pointer components.

Note: The procedure pointer `prc_get_id` triggers a bug in nagfor5.2(649) [incorrect C generated], apparently related to the string argument of this procedure. Fortunately, we can live without it.

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_t
```

⟨*Process libraries: types*⟩+≡
```
    type :: process_library_t
      ! private
      logical :: static = .false.
      integer :: status = STAT_UNKNOWN
      type(string_t) :: basename
      type(string_t) :: srcname
      type(string_t) :: libname
      integer :: n_prc = 0
      type(process_configuration_t), pointer :: prc_first => null ()
      type(process_configuration_t), pointer :: prc_last => null ()
      type(dlaccess_t) :: dlaccess
      procedure(prc_get_n_processes), nopass, pointer :: get_n_prc => null ()
      procedure(prc_get_stringptr), nopass, pointer :: get_process_id => null ()
      procedure(prc_get_stringptr), nopass, pointer :: get_model_name => null ()
      procedure(prc_get_stringptr), nopass, pointer :: &
           get_restrictions => null ()
      procedure(prc_get_stringptr), nopass, pointer :: get_md5sum => null ()
      procedure(prc_get_int), nopass, pointer :: get_n_in  => null ()
      procedure(prc_get_int), nopass, pointer :: get_n_out => null ()
      procedure(prc_get_int), nopass, pointer :: get_n_flv => null ()
```

```
      procedure(prc_get_int), nopass, pointer :: get_n_hel => null ()
      procedure(prc_get_int), nopass, pointer :: get_n_col => null ()
      procedure(prc_get_int), nopass, pointer :: get_n_cin => null ()
      procedure(prc_get_int), nopass, pointer :: get_n_cf  => null ()
      procedure(prc_set_int_tab1), nopass, pointer :: set_flv_state => null ()
      procedure(prc_set_int_tab1), nopass, pointer :: set_hel_state => null ()
      procedure(prc_set_int_tab2), nopass, pointer :: set_col_state => null ()
      procedure(prc_set_cf_tab), nopass, pointer :: set_cf_table => null ()
      procedure(prc_get_fptr), nopass, pointer :: init_get_fptr  => null()
      procedure(prc_get_fptr), nopass, pointer :: final_get_fptr => null()
      procedure(prc_get_fptr), nopass, pointer :: &
           update_alpha_s_get_fptr => null ()
      procedure(prc_get_fptr), nopass, pointer :: &
           reset_helicity_selection_get_fptr => null ()
      procedure(prc_get_fptr), nopass, pointer :: new_event_get_fptr => null ()
      procedure(prc_get_fptr), nopass, pointer :: is_allowed_get_fptr => null ()
      procedure(prc_get_fptr), nopass, pointer :: get_amplitude_get_fptr &
           => null ()
      procedure(prclib_unload_hook), nopass, pointer :: unload_hook => null ()
      procedure(prclib_reload_hook), nopass, pointer :: reload_hook => null ()
      type(process_library_t), pointer :: next => null ()
    end type process_library_t
```

Just allocate the configuration array and set filenames, the rest comes later.
Note that because libtool may be used, the actual `libname` can be determined
only after the library has been created.

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_init
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_init (prc_lib, name, os_data)
      type(process_library_t), intent(out) :: prc_lib
      type(string_t), intent(in) :: name
      type(os_data_t), intent(in) :: os_data
      prc_lib%basename = name
      prc_lib%srcname = name // os_data%fc_src_ext
      prc_lib%status = STAT_CONFIGURED
    end subroutine process_library_init
```

Delete the process configuration list, if any.

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_clear_configuration (prc_lib)
      type(process_library_t), intent(inout) :: prc_lib
      type(process_configuration_t), pointer :: current
      do while (associated (prc_lib%prc_first))
         current => prc_lib%prc_first
         prc_lib%prc_first => current%next
         deallocate (current)
      end do
      prc_lib%prc_last => null ()
      prc_lib%n_prc = 0
    end subroutine process_library_clear_configuration
```

943

Close the library access if it is open. Delete the process-configuration list.

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_final
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_final (prc_lib)
      type(process_library_t), intent(inout) :: prc_lib
      if (.not. prc_lib%static)  call dlaccess_final (prc_lib%dlaccess)
      call process_library_clear_configuration (prc_lib)
    end subroutine process_library_final
```

Given a pointer to a library, return the next pointer.

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_advance
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_advance (prc_lib)
      type(process_library_t), pointer :: prc_lib
      prc_lib => prc_lib%next
    end subroutine process_library_advance
```

Output (called by the 'show' command):

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_write
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_write (prc_lib, unit)
      type(process_library_t), intent(in) :: prc_lib
      integer, intent(in), optional :: unit
      type(string_t) :: status
      type(process_configuration_t), pointer :: current
      select case (prc_lib%status)
      case (STAT_UNKNOWN)
        status = "[unknown]"
      case (STAT_CONFIGURED)
        status = "[open]"
      case (STAT_CODE_GENERATED)
        status = "[generated code]"
      case (STAT_COMPILED)
        status = "[compiled]"
      case (STAT_LOADED)
        if (prc_lib%static) then
          status = "[static]"
        else
          status = "[loaded]"
        end if
      end select
      call msg_message ("Process library: " // char (prc_lib%basename) &
           // " " // char (status), unit)
      current => prc_lib%prc_first
      do while (associated (current))
        call process_configuration_write (current, unit)
        current => current%next
      end do
```

```
    end subroutine process_library_write
```

13.1.6   Accessing contents

Tell/set if the library is static or dynamic

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_set_static
    public :: process_library_is_static
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_set_static (prc_lib, flag)
      type(process_library_t), intent(inout) :: prc_lib
      logical, intent(in) :: flag
      prc_lib%static = flag
    end subroutine process_library_set_static

    function process_library_is_static (prc_lib) result (flag)
      logical :: flag
      type(process_library_t), intent(in) :: prc_lib
      flag = prc_lib%static
    end function process_library_is_static
```

Return the nominal compilation status of a library.

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_is_compiled
    public :: process_library_is_loaded
```

⟨*Process libraries: procedures*⟩+≡
```
    function process_library_is_compiled (prc_lib) result (flag)
      logical :: flag
      type(process_library_t), intent(in) :: prc_lib
      flag = prc_lib%status >= STAT_COMPILED
    end function process_library_is_compiled

    function process_library_is_loaded (prc_lib) result (flag)
      logical :: flag
      type(process_library_t), intent(in) :: prc_lib
      flag = prc_lib%status >= STAT_LOADED
    end function process_library_is_loaded
```

Return the name of a library (the basename).

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_get_name
```

⟨*Process libraries: procedures*⟩+≡
```
    function process_library_get_name (prc_lib) result (name)
      type(string_t) :: name
      type(process_library_t), intent(in) :: prc_lib
      name = prc_lib%basename
    end function process_library_get_name
```

945

Return the number of processes defined so far.

⟨Process libraries: public⟩+≡
```
    public :: process_library_get_n_processes
```

⟨Process libraries: procedures⟩+≡
```
    function process_library_get_n_processes (prc_lib) result (n)
      integer :: n
      type(process_library_t), intent(in) :: prc_lib
      n = prc_lib%n_prc
    end function process_library_get_n_processes
```

Return the pointer to a process with specified tag.

⟨Process libraries: procedures⟩+≡
```
    function process_library_get_process_ptr (prc_lib, prc_id) result (current)
      type(process_library_t), intent(in), target :: prc_lib
      type(string_t), intent(in) :: prc_id
      type(process_configuration_t), pointer :: current
      current => prc_lib%prc_first
      do while (associated (current))
         if (current%id == prc_id)  return
         current => current%next
      end do
    end function process_library_get_process_ptr
```

Return the pointer to a process with specified tag.

⟨Process libraries: public⟩+≡
```
    public :: process_library_check_name_consistency
```

⟨Process libraries: procedures⟩+≡
```
    subroutine process_library_check_name_consistency (prc_id, prc_lib)
      type(process_library_t), intent(in), target :: prc_lib
      type(string_t), intent(in) :: prc_id
      if (char (prc_id) == 'prc') &
           call msg_fatal ("The name 'prc' cannot " // &
                    "be chosen as a valid process name.")
      if (prc_id == prc_lib%basename) &
           call msg_fatal ("Process and library names must not be identical ('" &
                    // char (prc_id) // "').")
    end subroutine process_library_check_name_consistency
```

Return the index of a process with specified tag. If the process is not found,
return zero.

⟨Process libraries: public⟩+≡
```
    public :: process_library_get_process_index
```

⟨Process libraries: procedures⟩+≡
```
    function process_library_get_process_index (prc_lib, prc_id) result (index)
      integer :: index
      type(process_library_t), intent(in), target :: prc_lib
      type(string_t), intent(in) :: prc_id
      type(process_configuration_t), pointer :: current
      index = 0
      current => prc_lib%prc_first
```

946

```
        do while (associated (current))
            index = index + 1
            if (current%id == prc_id)  return
            current => current%next
        end do
        index = 0
    end function process_library_get_process_index
```

13.1.7 Creating a process library

Configure a specific process in the list. First check if the process exists, then either edit the existing process configuration or initiate a new one. If a status is given, mark the process configuration accordingly. Overwrite any existing configuration for the given process ID. If the `rebuild_library` flag is set, do this silently and reset the status. If it is absent or unset, we want to keep the old configuration as far as possible. If the checksum has changed issue a warning that the configuration was overwritten. If the old status was higher, keep it.

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_append
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_append &
          (prc_lib, prc_id, model, prt_in, prt_out, method, &
           status, restrictions, rebuild_library, message, known_md5sum)
      type(process_library_t), intent(inout), target :: prc_lib
      type(string_t), intent(in) :: prc_id
      type(model_t), intent(in), target :: model
      type(string_t), dimension(:), intent(in) :: prt_in, prt_out
      integer, intent(in), optional :: status, method
      type(string_t), intent(in), optional :: restrictions
      logical, intent(in), optional :: rebuild_library, message
      character(32), intent(in), optional :: known_md5sum
      type(process_configuration_t), pointer :: current
      character(32) :: old_md5sum
      integer :: old_status
      integer :: old_method
      logical :: keep_status
      logical :: msg
      keep_status = .true.;  if (present (rebuild_library))  keep_status = .not. rebuild_library
      msg = .false.;  if (present (message))  msg = message
      current => process_library_get_process_ptr (prc_lib, prc_id)
      if (associated (current)) then
         old_md5sum = current%md5sum
         old_status = current%status
         old_method = current%method
         call process_configuration_init &
              (current, prc_id, model, prt_in, prt_out, old_method, status, &
               restrictions, known_md5sum)
         if (size (prt_in) == 0) then
            call msg_warning ("Process '" // char (prc_id) &
                 // "': matrix element vanishes in selected model '" &
                 // char (model_get_name (model)) // "'")
```

```
            else if (keep_status) then
                if (current%md5sum == old_md5sum) then
                    if (current%status <= old_status) then
                        call msg_message ("Process '" // char (prc_id) &
                            // "': keeping configuration")
                        current%status = old_status
                    else
                        call msg_message ("Process '" // char (prc_id) &
                            // "': updating configuration")
                    end if
                else
                    call msg_warning ("Process '" // char (prc_id) &
                        // "': configuration changed, overwriting.")
                end if
            else
                if (current%md5sum /= old_md5sum) then
                    call msg_message ("Process '" // char (prc_id) &
                        // "': ignoring previous configuration")
                end if
            end if
        else
            allocate (current)
            if (associated (prc_lib%prc_last)) then
                prc_lib%prc_last%next => current
            else
                prc_lib%prc_first => current
            end if
            prc_lib%prc_last => current
            prc_lib%n_prc = prc_lib%n_prc + 1
            call process_library_check_name_consistency (prc_id, prc_lib)
            call process_configuration_init &
                (current, prc_id, model, prt_in, prt_out, method, status, &
                 restrictions, known_md5sum)
            call process_update_code_status (current, keep_status)
            if (msg)  call msg_message &
                ("Added process to library '" // char (prc_lib%basename) // "':")
        end if
        if (msg)  call process_configuration_write (current)
    end subroutine process_library_append
```

Look for an existing file for the current process and its MD5 signature. If
successful and a rebuild flag is set, reset the status to STAT_CODE_GENERATED.
Otherwise, just issue appropriate diagnostic messages.

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_update_code_status (prc_conf, keep_status)
      type(process_configuration_t), intent(inout) :: prc_conf
      logical, intent(in) :: keep_status
      type(string_t) :: filename
      logical :: exist, found
      integer :: u, iostat
      character(80) :: buffer
      character(32) :: md5sum
      filename = prc_conf%id // ".f90"
```

```
        inquire (file=char(filename), exist=exist)
        if (exist) then
            found = .false.
            u = free_unit ()
            open (u, file=char(filename), action="read")
            SCAN_FILE: do
                read (u, "(A)", iostat=iostat)  buffer
                select case (iostat)
                case (0)
                    select case (buffer(1:12))
                    case ("    md5sum =")
                        md5sum = buffer(15:47)
                        if (keep_status) then
                            if (prc_conf%status < STAT_CODE_GENERATED) then
                                if (md5sum == prc_conf%md5sum) then
                                    call msg_message ("Process '" // char (prc_conf%id) &
                                         // "': using existing source code")
                                    prc_conf%status = STAT_CODE_GENERATED
                                else
                                    call msg_warning ("Process '" // char (prc_conf%id) &
                                         // "': will overwrite existing source code")
                                end if
                            else if (md5sum /= prc_conf%md5sum) then
                                call msg_warning ("Process '" // char (prc_conf%id) &
                                     // "': source code and loaded checksums differ")
                            end if
                        else if (prc_conf%status < STAT_CODE_GENERATED) then
                            call msg_message ("Process '" // char (prc_conf%id) &
                                 // "': ignoring existing source code")
                        end if
                        found = .true.
                        exit SCAN_FILE
                    end select
                case default
                    exit SCAN_FILE
                end select
            end do SCAN_FILE
            close (u)
            if (.not. found) &
                 call msg_warning ("Process '" // char (prc_conf%id) &
                 // "': No MD5 sum found in source code")
        end if
    end subroutine process_update_code_status
```

Check whether all processes in the current library are configured, compiled and
loaded, and update the library status accordingly.

   If the library needs recompilation, unload it now if necessary.

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_update_status
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_update_status (prc_lib)
      type(process_library_t), intent(inout), target :: prc_lib
      type(process_configuration_t), pointer :: prc_conf
```

```
      integer :: initial_status
      initial_status = prc_lib%status
      prc_conf => prc_lib%prc_first
      do while (associated (prc_conf))
         prc_lib%status = min (prc_lib%status, prc_conf%status)
         prc_conf => prc_conf%next
      end do
      if (initial_status == STAT_LOADED .and. prc_lib%status < STAT_LOADED) &
           call process_library_unload (prc_lib)
    end subroutine process_library_update_status
```

Recover process configuration from a loaded library. Existing configurations for processes present in the loaded library will be overwritten. Return the pointer to the model appropriate for the loaded library.

⟨*Process libraries: procedures*⟩+≡

```
    subroutine process_library_load_configuration &
         (prc_lib, os_data, model)
      type(process_library_t), intent(inout), target :: prc_lib
      type(os_data_t), intent(in) :: os_data
      type(model_t), pointer :: model
      integer :: n_prc, p, n_flv, n_in, n_out, n_tot, i
      integer(c_int) :: pid
      integer, dimension(:,:), allocatable :: flv_state
      integer(c_int), dimension(:,:), allocatable, target :: flv_state_tmp
      type(string_t) :: prc_id, model_name, filename, restrictions
      type(string_t), dimension(:), allocatable :: prt_in, prt_out
      character(32) :: md5sum
      n_prc = prc_lib% get_n_prc ()
      SCAN_PROCESSES: do p = 1, n_prc
         pid = p
         prc_id = process_library_get_process_id (prc_lib, pid)
         md5sum = process_library_get_process_md5sum (prc_lib, pid)
         model_name = process_library_get_process_model_name (prc_lib, pid)
         restrictions = process_library_get_process_restrictions (prc_lib, pid)
         filename = model_name // ".mdl"
         model => null ()
         call model_list_read_model (model_name, filename, os_data, model)
         if (.not. associated (model)) then
            call msg_error ("Process library '" // char (prc_lib%basename) &
                 // "', process '" // char (prc_id) // "': " &
                 // "model unavailable, process skipped")
            cycle SCAN_PROCESSES
         end if
         n_in  = prc_lib% get_n_in  (pid)
         n_out = prc_lib% get_n_out (pid)
         n_tot = n_in + n_out
         n_flv = prc_lib% get_n_flv (pid)
         allocate (flv_state (n_tot, n_flv))
         allocate (flv_state_tmp (n_tot, n_flv))
         allocate (prt_in  (n_in ))
         allocate (prt_out (n_out))
         call prc_lib% set_flv_state (pid, &
              c_loc (flv_state_tmp), &
```

```
                int((/n_tot, n_flv/), kind=c_int))
          flv_state = flv_state_tmp
          do i = 1, n_in
             prt_in(i) = particle_name_string (flv_state (i, :), model)
          end do
          do i = 1, n_out
             prt_out(i) = particle_name_string (flv_state (n_in+i, :), model)
          end do
          call process_library_append &
               (prc_lib, prc_id, model, prt_in, prt_out, &
                status=STAT_LOADED, restrictions=restrictions, &
                known_md5sum=md5sum)
          deallocate (prt_in, prt_out, flv_state, flv_state_tmp)
       end do SCAN_PROCESSES
    contains
      function particle_name_string (ff, model) result (prt)
        type(string_t) :: prt
        integer, dimension(:), intent(in) :: ff
        type(model_t), intent(in), target :: model
        type(flavor_t) :: flv
        integer :: i
        prt = ""
        do i = 1, size (ff)
           if (all (ff(i) /= ff(:i-1))) then
              call flavor_init (flv, ff(i), model)
              if (prt /= "")  prt = prt // ":"
              prt = prt // flavor_get_name (flv)
           end if
        end do
      end function particle_name_string
    end subroutine process_library_load_configuration
```

⟨*Process libraries: public*⟩+≡

```
    public :: process_library_get_process_id
    public :: process_library_get_process_pid
    public :: process_library_get_process_md5sum
    public :: process_library_get_process_model_name
```

⟨*Process libraries: procedures*⟩+≡

```
    function process_library_get_process_id (prc_lib, pid) result (process_id)
      type(string_t) :: process_id
      type(process_library_t), intent(in), target :: prc_lib
      integer(c_int), intent(in) :: pid
      type(c_ptr) :: cptr
      integer(c_int) :: len
      character(kind=c_char), dimension(:), pointer :: char_array
      integer, dimension(1) :: shape
      call prc_lib% get_process_id (pid, cptr, len)
      if (c_associated (cptr)) then
         shape(1) = len
         call c_f_pointer (cptr, char_array, shape)
         process_id = char_from_array (char_array)
         call prc_lib% get_process_id (0_c_int, cptr, len)
      else
```

```
      process_id = ""
    end if
end function process_library_get_process_id

function process_library_get_process_pid (prc_lib, id) result (process_pid)
    type(process_library_t), intent(in) :: prc_lib
    type(string_t), intent(in) :: id
    integer :: process_pid, pid, n_proc
    process_pid = -1
    n_proc = process_library_get_n_processes (prc_lib)
    if (n_proc <= 0) return
    do pid = 1, n_proc
      if (process_library_get_process_id (prc_lib, pid) == id) then
        process_pid = pid
        return
      end if
    end do
end function process_library_get_process_pid

function process_library_get_process_model_name &
     (prc_lib, pid) result (model_name)
  type(string_t) :: model_name
  type(process_library_t), intent(in), target :: prc_lib
  integer(c_int), intent(in) :: pid
  type(c_ptr) :: cptr
  integer(c_int) :: len
  character(kind=c_char), dimension(:), pointer :: char_array
  integer, dimension(1) :: shape
  call prc_lib% get_model_name (pid, cptr, len)
  if (c_associated (cptr)) then
    shape(1) = len
    call c_f_pointer (cptr, char_array, shape)
    model_name = char_from_array (char_array)
    call prc_lib% get_model_name (0_c_int, cptr, len)
  else
    model_name = ""
  end if
end function process_library_get_process_model_name

function process_library_get_process_restrictions &
     (prc_lib, pid) result (restrictions)
  type(string_t) :: restrictions
  type(process_library_t), intent(in), target :: prc_lib
  integer(c_int), intent(in) :: pid
  type(c_ptr) :: cptr
  integer(c_int) :: len
  character(kind=c_char), dimension(:), pointer :: char_array
  integer, dimension(1) :: shape
  call prc_lib% get_restrictions (pid, cptr, len)
  if (c_associated (cptr)) then
    shape(1) = len
    call c_f_pointer (cptr, char_array, shape)
    restrictions = char_from_array (char_array)
    call prc_lib% get_restrictions (0_c_int, cptr, len)
```

```
      else
         restrictions = ""
      end if
   end function process_library_get_process_restrictions

   function process_library_get_process_md5sum (prc_lib, pid) result (md5sum)
      type(string_t) :: md5sum
      type(process_library_t), intent(in), target :: prc_lib
      integer(c_int), intent(in) :: pid
      type(c_ptr) :: cptr
      integer(c_int) :: len
      character(kind=c_char), dimension(:), pointer :: char_array
      integer, dimension(1) :: shape
      call prc_lib% get_md5sum (pid, cptr, len)
      if (c_associated (cptr)) then
         shape(1) = len
         call c_f_pointer (cptr, char_array, shape)
         md5sum = char_from_array (char_array)
         call prc_lib% get_md5sum (0_c_int, cptr, len)
      else
         md5sum = ""
      end if
   end function process_library_get_process_md5sum
```

Auxiliary: Transform a character array into a character string.

⟨*Process libraries: procedures*⟩+≡
```
   function char_from_array (a) result (char)
      character(kind=c_char), dimension(:), intent(in) :: a
      character(len=size(a)) :: char
      integer :: i
      do i = 1, len (char)
         char(i:i) = a(i)
      end do
   end function char_from_array
```

Generate process source code. Do this for all processes which have just been configured, unless there is a source-code file with identical MD5sum.

⟨*Process libraries: public*⟩+≡
```
   public :: process_library_generate_code
```

⟨*Process libraries: procedures*⟩+≡
```
   subroutine process_library_generate_code (prc_lib, os_data, simulate)
      type(process_library_t), intent(in) :: prc_lib
      type(os_data_t), intent(in) :: os_data
      logical, intent(in), optional :: simulate
      type(process_configuration_t), pointer :: current
      integer :: status
      call msg_message ("Generating code for process library '" &
           // char (process_library_get_name (prc_lib)) // "'")
      current => prc_lib%prc_first
      SCAN_PROCESSES: do while (associated (current))
         select case (current%status)
         case (STAT_CONFIGURED)
```

```
                select case (current%method)
                case (PRC_OMEGA)
                   call call_omega (current, os_data, status, simulate)
                   if (status == 0) then
                      current%status = STAT_CODE_GENERATED
                   else
                      call msg_error ("Process '" // char (current%id) &
                            // "': code generation failed")
                   end if
                case (PRC_TEST)
                   call write_unit_matrix_element (current, os_data, status, unit=.false.)
                   if (status == 0) then
                      current%status = STAT_CODE_GENERATED
                   else
                      call msg_error ("Process '" // char (current%id) &
                            // "': code generation failed")
                   end if
                case (PRC_UNIT)
                   call write_unit_matrix_element (current, os_data, status, unit=.true.)
                   if (status == 0) then
                      current%status = STAT_CODE_GENERATED
                   else
                      call msg_error ("Process '" // char (current%id) &
                            // "': code generation failed")
                   end if
                case default
                   call msg_fatal ("These methods are not yet implemented.")
                end select
             case (STAT_CODE_GENERATED:)
                call msg_message ("Skipping process '" // char (current%id) &
                      // "' (source code exists)")
             case default
                call msg_message ("Skipping process '" // char (current%id) &
                      // "' (undefined configuration)")
             end select
             current => current%next
          end do SCAN_PROCESSES
       end subroutine process_library_generate_code
```

Call O'MEGA for process-code generation.

⟨*Process libraries: procedures*⟩+≡
```
    subroutine call_omega (prc_conf, os_data, status, simulate)
      type(process_configuration_t), intent(in) :: prc_conf
      type(os_data_t), intent(in) :: os_data
      integer, intent(out) :: status
      logical, intent(in), optional :: simulate
      type(string_t) :: command_string, binary_name
      type(string_t) :: model_id, omega_mode, omega_cascade
      integer :: j
      logical :: sim, binary_found
      sim = .false.;  if (present (simulate))  sim = simulate
      call msg_message ("Calling O'Mega for process '" &
            // char (prc_conf%id) // "'")
```

```
model_id = model_get_name (prc_conf%model)
binary_name = "omega_" // model_id // ".opt"
binary_found = .false.
if (.not. os_data%use_testfiles) then
   command_string = os_data%whizard_omega_binpath_local &
      // "/" // binary_name
   inquire (file=char (command_string), exist=binary_found)
end if
if (.not. binary_found) then
   command_string = os_data%whizard_omega_binpath // "/" // binary_name
   inquire (file=char (command_string), exist=binary_found)
end if
if (.not. binary_found) &
   call msg_fatal ("O'Mega binary """ // char (binary_name) // """ not found")
select case (prc_conf%n_in)
case (1);  omega_mode = "-decay"
case (2);  omega_mode = "-scatter"
end select
if (prc_conf%restrictions == "") then
   omega_cascade = ""
else
   omega_cascade = " -cascade '" // prc_conf%restrictions // "'"
end if
command_string = command_string &
     // " -o " // prc_conf%id // ".f90" &
     // " -target:whizard" &
     // " -target:parameter_module parameters_" // model_id &
     // " -target:module " // prc_conf%id &
     // " -target:md5sum " // prc_conf%md5sum &
     // omega_cascade &
     // " -fusion:progress" &
     // " " // omega_mode
command_string = command_string // " "
do j = 1, prc_conf%n_in
   if (j == 1) then
       command_string = command_string // "'"
   else
       command_string = command_string // " "
   end if
   command_string = command_string // prc_conf%prt_in(j)
end do
command_string = command_string // " ->"
do j = 1, prc_conf%n_out
   command_string = command_string &
        // " " // prc_conf%prt_out(j)
end do
command_string = command_string // "'"
if (sim) then
   command_string = "cp " // os_data%whizard_testdatapath // "/" &
        // prc_conf%id // ".f90 ."
   call msg_message ("[call not executed, instead: copy file from " &
        // char (os_data%whizard_testdatapath) // "]")
end if
call os_system_call (command_string, status, verbose=.true.)
```

```
      end subroutine call_omega
```

Intrinsic **WHIZARD** call for unit matrix elements generation. Note that color flows, spins and flavors are taken as trivial, i.e. there is one (trivial) color flow, one spin combination for all particles, and tensor products for flavors are ignored.

⟨*Process libraries: procedures*⟩+≡

```
      subroutine write_unit_matrix_element (prc_conf, os_data, status, unit)
        type(process_configuration_t), intent(in) :: prc_conf
        type(os_data_t), intent(in) :: os_data
        integer, intent(out) :: status
        logical, intent(in) :: unit
        integer, dimension(prc_conf%n_in) :: prt_in, mult_in
        type(flavor_t), dimension(1:prc_conf%n_in) :: flv_in
        integer, dimension(prc_conf%n_out) :: prt_out, mult_out
        integer, dimension(prc_conf%n_tot) :: prt, mult
        integer, dimension(:,:), allocatable :: sxxx
        integer :: dummy
        type(flavor_t), dimension(1:prc_conf%n_out) :: flv_out
        type(string_t) :: proc_str, comment_str
        integer :: u, i, j, count
        integer :: hel, hel_in, hel_out, fac, factor
        type(string_t) :: filename
        comment_str = ""
        do i = 1, prc_conf%n_in
           comment_str = comment_str // prc_conf%prt_in(i) // " "
        end do
        do j = 1, prc_conf%n_out
           comment_str = comment_str // prc_conf%prt_out(j) // " "
        end do
        do i = 1, prc_conf%n_in
           prt_in(i) = model_get_particle_pdg (prc_conf%model, prc_conf%prt_in(i))
           call flavor_init (flv_in(i), prt_in(i), prc_conf%model)
           mult_in(i) = flavor_get_multiplicity (flv_in(i))
           mult(i) = mult_in(i)
           end do
        do j = 1, prc_conf%n_out
           prt_out(j) = model_get_particle_pdg (prc_conf%model, prc_conf%prt_out(j))
           call flavor_init (flv_out(j), prt_out(j), prc_conf%model)
           mult_out(j) = flavor_get_multiplicity (flv_out(j))
           mult(prc_conf%n_in + j) = mult_out(j)
           end do
        prt(1:prc_conf%n_in) = prt_in(1:prc_conf%n_in)
        prt(prc_conf%n_in+1:prc_conf%n_tot) = prt_out(1:prc_conf%n_out)
        proc_str = converter (prt)
        hel_in = product (mult_in)
        hel_out = product (mult_out)
        hel = hel_in * hel_out
        fac = hel
        dummy = 1
        factor = 1
        if (prc_conf%n_out >= 3) then
           do i = 3, prc_conf%n_out
              factor = factor * (i - 2) * (i - 1)
```

```
      end do
end if
allocate (sxxx(1:hel,1:prc_conf%n_tot))
call create_spin_table (dummy,hel,fac,mult,sxxx)
call msg_message ("Writing test matrix element for process '" &
     // char (prc_conf%id) // "'")
filename = prc_conf%id // ".f90"
u = free_unit ()
open (unit=u, file=char(filename), action="write")
write (u, "(A)") "! File generated automatically by WHIZARD"
write (u, "(A)") "!                                          "
write (u, "(A)") "! Note that irresp. of what you demanded WHIZARD"
write (u, "(A)") "! treats this as colorless process         "
write (u, "(A)") "!                                          "
write (u, "(A)") "module " // char(prc_conf%id)
write (u, "(A)") "                                           "
write (u, "(A)") "  use kinds"
write (u, "(A)") "  use omega_color, OCF => omega_color_factor"
write (u, "(A)") "                                           "
write (u, "(A)") "  implicit none"
write (u, "(A)") "  private"
write (u, "(A)") "                                           "
write (u, "(A)") "  public :: md5sum"
write (u, "(A)") "  public :: number_particles_in, number_particles_out"
write (u, "(A)") "  public :: number_spin_states, spin_states"
write (u, "(A)") "  public :: number_flavor_states, flavor_states"
write (u, "(A)") "  public :: number_color_flows, color_flows"
write (u, "(A)") "  public :: number_color_indices, number_color_factors, &"
write (u, "(A)") "     color_factors, color_sum"
write (u, "(A)") "  public :: init, final"
write (u, "(A)") "  public :: reset_helicity_selection"
write (u, "(A)") "                                           "
write (u, "(A)") "  public :: new_event, is_allowed, get_amplitude"
write (u, "(A)") "        "
write (u, "(A)") "  real(default), parameter :: &"
write (u, "(A)") "       & conv = 0.38937966e12_default"
write (u, "(A)") "       "
write (u, "(A)") "  real(default), parameter :: &"
write (u, "(A)") "       & pi = 3.1415926535897932384626433832795028841972_default"
write (u, "(A)") "       "
write (u, "(A)") "  real(default), parameter :: &"
if (unit) then
   write (u, "(A)") "                      & const = 1"
else
   write (u, "(A,1x,I0,A)") "        & const = (16 * pi / conv) * " &
      // "(16 * pi**2)**(", prc_conf%n_out, "-2) "
end if
write (u, "(A)") "        "
write (u, "(A,1x,I0)") "  integer, parameter, private :: n_prt =  ", &
   prc_conf%n_tot
write (u, "(A,1x,I0)") "  integer, parameter, private :: n_in = ", &
   prc_conf%n_in
write (u, "(A,1x,I0)") "  integer, parameter, private :: n_out = ", &
   prc_conf%n_out
```

```
write (u, "(A)") " integer, parameter, private :: n_cflow = 1"
write (u, "(A)") " integer, parameter, private :: n_cindex = 2"
write (u, "(A)") " !!! We ignore tensor products and take only one flavor state."
write (u, "(A)") " integer, parameter, private :: n_flv = 1"
write (u, "(A,1x,I0)") " integer, parameter, private :: n_hel = ", hel
write (u, "(A)") "                                                           "
write (u, "(A)") " logical, parameter, private :: T = .true."
write (u, "(A)") " logical, parameter, private :: F = .false."
write (u, "(A)") "                                                           "
do i = 1, hel
   write (u, "(A)") " integer, dimension(n_prt), parameter, private :: &"
   write (u, "(A)") "    " // s_conv(i) // " = (/ " // char(converter(sxxx(i,1:prc_conf%n_tot)
end do
write (u, "(A)") " integer, dimension(n_prt,n_hel), parameter, private :: table_spin_states =
write (u, "(A)") "    reshape ( (/ & "
do i = 1, hel-1
   write (u, "(A)") "                    " // s_conv(i) // ", & "
end do
write (u, "(A)") "                    " // s_conv(hel) // " & "
write (u, "(A)") "                 /), (/ n_prt, n_hel /) )"
write (u, "(A)") "                                                             "
write (u, "(A)") " integer, dimension(n_prt), parameter, private :: &"
write (u, "(A)") "    f0001 = (/ " // char(proc_str) // " /)   ! " // char(comment_str)
write (u, "(A)") " integer, dimension(n_prt,n_flv), parameter, private :: table_flavor_states
write (u, "(A)") "    reshape ( (/ f0001 /), (/ n_prt, n_flv /) )"
write (u, "(A)") "                                                             "
write (u, "(A)") " integer, dimension(n_cindex, n_prt), parameter, private :: &"
write (u, "(A)") "    c0001 = reshape ( (/ " // (repeat ("0,0, ", prc_conf%n_tot-1)) &
write (u, "(A)") "                 // "0,0 /), " // " (/ n_cindex, n_prt /) )"
write (u, "(A)") " integer, dimension(n_cindex, n_prt, n_cflow), parameter, private :: &"
write (u, "(A)") "    table_color_flows = reshape ( (/ c0001 /), (/ n_cindex, n_prt, n_cflow /)
write (u, "(A)") "                                                             "
write (u, "(A)") " logical, dimension(n_prt), parameter, private :: & "
write (u, "(A)") "    g0001 = (/ "  // (repeat ("F, ", prc_conf%n_tot-1)) // "F /) "
write (u, "(A)") " logical, dimension(n_prt, n_cflow), parameter, private :: table_ghost_flag
write (u, "(A)") "    reshape ( (/ g0001 /), (/ n_prt, n_cflow /) )"
write (u, "(A)") "                                                             "
write (u, "(A)") " integer, parameter, private :: n_cfactors = 1"
write (u, "(A)") " type(OCF), dimension(n_cfactors), parameter, private :: &"
write (u, "(A)") "    table_color_factors = (/  OCF(1,1,+1._default) /)"
write (u, "(A)") "                                                             "
write (u, "(A)") " logical, dimension(n_flv), parameter, private :: a0001 = (/ T /)"
write (u, "(A)") " logical, dimension(n_flv, n_cflow), parameter, private :: &"
write (u, "(A)") "    flv_col_is_allowed = reshape ( (/ a0001 /), (/ n_flv, n_cflow /) )"
write (u, "(A)") "                                                             "
write (u, "(A)") " complex(default), dimension (n_flv, n_hel, n_cflow), private, save :: amp"
write (u, "(A)") "                                                             "
write (u, "(A)") " logical, dimension(n_hel), private, save :: hel_is_allowed = T"
write (u, "(A)") "                                                             "
write (u, "(A)") "contains"
write (u, "(A)") "                                                             "
write (u, "(A)") " pure function md5sum ()"
write (u, "(A)") "    character(len=32) :: md5sum"
write (u, "(A)") "    ! DON'T EVEN THINK of modifying the following line!"
```

```
          write (u, "(A)") "    md5sum = """ // prc_conf%md5sum // """"
          write (u, "(A)") "  end function md5sum"
          write (u, "(A)") "                                                   "
          write (u, "(A)") "  subroutine init (par)"
          write (u, "(A)") "    real(default), dimension(*), intent(in) :: par"
          write (u, "(A)") "  end subroutine init"
          write (u, "(A)") "                                                   "
          write (u, "(A)") "  subroutine final ()"
          write (u, "(A)") "  end subroutine final"
          write (u, "(A)") "                                                   "
          write (u, "(A)") "  pure function number_particles_in () result (n)"
          write (u, "(A)") "    integer :: n"
          write (u, "(A)") "    n = n_in"
          write (u, "(A)") "  end function number_particles_in"
          write (u, "(A)") "                                                   "
          write (u, "(A)") "  pure function number_particles_out () result (n)"
          write (u, "(A)") "    integer :: n"
          write (u, "(A)") "    n = n_out"
          write (u, "(A)") "  end function number_particles_out"
          write (u, "(A)") "                                                   "
          write (u, "(A)") "  pure function number_spin_states () result (n)"
          write (u, "(A)") "    integer :: n"
          write (u, "(A)") "    n = size (table_spin_states, dim=2)"
          write (u, "(A)") "  end function number_spin_states"
          write (u, "(A)") "                                                   "
          write (u, "(A)") "  pure subroutine spin_states (a)"
          write (u, "(A)") "    integer, dimension(:,:), intent(out) :: a"
          write (u, "(A)") "    a = table_spin_states"
          write (u, "(A)") "  end subroutine spin_states"
          write (u, "(A)") "                                                   "
          write (u, "(A)") "  pure function number_flavor_states () result (n)"
          write (u, "(A)") "    integer :: n"
          write (u, "(A)") "    n = 1"
          write (u, "(A)") "  end function number_flavor_states"
          write (u, "(A)") "                                                   "
          write (u, "(A)") "  pure subroutine flavor_states (a)"
          write (u, "(A)") "    integer, dimension(:,:), intent(out) :: a"
          write (u, "(A)") "    a = table_flavor_states"
          write (u, "(A)") "  end subroutine flavor_states"
          write (u, "(A)") "                                                   "
          write (u, "(A)") "  pure function number_color_indices () result (n)"
          write (u, "(A)") "    integer :: n"
          write (u, "(A)") "    n = size(table_color_flows, dim=1)"
          write (u, "(A)") "  end function number_color_indices"
          write (u, "(A)") "                                                   "
          write (u, "(A)") "  pure subroutine color_factors (cf)"
          write (u, "(A)") "    type(OCF), dimension(:), intent(out) :: cf"
          write (u, "(A)") "    cf = table_color_factors"
          write (u, "(A)") "  end subroutine color_factors"
          write (u, "(A)") "                                                   "
          write (u, "(A)") "  pure function color_sum (flv, hel) result (amp2)"
          write (u, "(A)") "    integer, intent(in) :: flv, hel"
          write (u, "(A)") "    real(kind=default) :: amp2"
          write (u, "(A)") "    amp2 = real (omega_color_sum (flv, hel, amp, table_color_factors))"
```

959

```
write (u, "(A)") "  end function color_sum"
write (u, "(A)") "                                                "
write (u, "(A)") "  pure function number_color_flows () result (n)"
write (u, "(A)") "    integer :: n"
write (u, "(A)") "    n = size (table_color_flows, dim=3)"
write (u, "(A)") "  end function number_color_flows"
write (u, "(A)") "                                                "
write (u, "(A)") "  pure subroutine color_flows (a, g)"
write (u, "(A)") "    integer, dimension(:,:,:), intent(out) :: a"
write (u, "(A)") "    logical, dimension(:,:), intent(out) :: g"
write (u, "(A)") "    a = table_color_flows"
write (u, "(A)") "    g = table_ghost_flags"
write (u, "(A)") "  end subroutine color_flows"
write (u, "(A)") "                                                "
write (u, "(A)") "  pure function number_color_factors () result (n)"
write (u, "(A)") "    integer :: n"
write (u, "(A)") "    n = size (table_color_factors)"
write (u, "(A)") "  end function number_color_factors"
write (u, "(A)") "                                                "
write (u, "(A)") "  subroutine new_event (p)"
write (u, "(A)") "    real(default), dimension(0:3,*), intent(in) :: p"
write (u, "(A)") "    call calculate_amplitudes (amp, p)"
write (u, "(A)") "  end subroutine new_event"
write (u, "(A)") "                                                "
write (u, "(A)") "  subroutine reset_helicity_selection (threshold, cutoff)"
write (u, "(A)") "    real(default), intent(in) :: threshold"
write (u, "(A)") "    integer, intent(in) :: cutoff"
write (u, "(A)") "  end subroutine reset_helicity_selection"
write (u, "(A)") "                                                "
write (u, "(A)") "  pure function is_allowed (flv, hel, col) result (yorn)"
write (u, "(A)") "    logical :: yorn"
write (u, "(A)") "    integer, intent(in) :: flv, hel, col"
write (u, "(A)") "    yorn = hel_is_allowed(hel) .and. flv_col_is_allowed(flv,col)"
write (u, "(A)") "  end function is_allowed"
write (u, "(A)") "                                                "
write (u, "(A)") "  pure function get_amplitude (flv, hel, col) result (amp_result)"
write (u, "(A)") "    complex(default) :: amp_result"
write (u, "(A)") "    integer, intent(in) :: flv, hel, col"
write (u, "(A)") "    amp_result = amp (flv, hel, col)"
write (u, "(A)") "  end function get_amplitude"
write (u, "(A)") "                                                "
write (u, "(A)") "  pure subroutine calculate_amplitudes (amp, k)"
write (u, "(A)") "    complex(default), dimension(:,:,:), intent(out) :: amp"
write (u, "(A)") "    real(default), dimension(0:3,*), intent(in) :: k"
write (u, "(A)") "    real(default) :: fac"
write (u, "(A)") "    integer :: i"
write (u, "(A)") "    ! We give all helicities the same weight!"
if (unit) then
   write (u, "(A)") "    amp = const"
else
   write (u, "(A,1x,I0,1x,A)") "    fac = ", factor
   write (u, "(A)") "    amp = sqrt((2 * (k(0,1)*k(0,2) &"
   write (u, "(A,1x,I0,A)") "         - dot_product (k(1:,1), k(1:,2)))) ** (3-", &
                        prc_conf%n_out, ")) * sqrt(const * fac)"
```

960

```fortran
        end if
        write (u, "(A,1x,I0,A)") "    amp = amp / sqrt(", hel_out, "._default)"
        write (u, "(A)") "  end subroutine calculate_amplitudes"
        write (u, "(A)") "                                                   "
        write (u, "(A)") "end module " // char(prc_conf%id)
        close (u, iostat=status)
        deallocate (sxxx)
  contains
    function s_conv (num) result (chrt)
      integer, intent(in) :: num
      character(len=10) :: chrt
      write (chrt, "(I10)") num
      chrt = trim(adjustl(chrt))
      if (num < 10) then
         chrt = "s000" // chrt
      else if (num < 100) then
         chrt = "s00" // chrt
      else if (num < 1000) then
         chrt = "s0" // chrt
      else
         chrt = "s" // chrt
      end if
    end function s_conv
    function converter (flv) result (str)
      integer, dimension(:), intent(in) :: flv
      type(string_t) :: str
      character(len=150), dimension(size(flv)) :: chrt
      integer :: i
      str = ""
      do i = 1, size(flv) - 1
         write (chrt(i), "(I10)") flv(i)
         str = str // var_str(trim(adjustl(chrt(i)))) // ", "
      end do
      write (chrt(size(flv)), "(I10)") flv(size(flv))
      str = str // trim(adjustl(chrt(size(flv))))
    end function converter
    integer function sj (j,m)
      integer, intent(in) :: j, m
      if (((j == 1) .and. (m == 1)) .or. &
          ((j == 2) .and. (m == 2)) .or. &
          ((j == 3) .and. (m == 3)) .or. &
          ((j == 4) .and. (m == 3)) .or. &
          ((j == 5) .and. (m == 4))) then
         sj = 1
      else if (((j == 2) .and. (m == 1)) .or. &
          ((j == 3) .and. (m == 1)) .or. &
          ((j == 4) .and. (m == 2)) .or. &
          ((j == 5) .and. (m == 2))) then
         sj = -1
      else if (((j == 3) .and. (m == 2)) .or. &
          ((j == 5) .and. (m == 3))) then
         sj = 0
      else if (((j == 4) .and. (m == 1)) .or. &
          ((j == 5) .and. (m == 1))) then
```

```
         sj = -2
      else if (((j == 4) .and. (m == 4)) .or. &
           ((j == 5) .and. (m == 5))) then
         sj = 2
      else
         call msg_fatal ("Write_unit_matrix_element: Wrong spin type")
      end if
    end function sj
    recursive subroutine create_spin_table (index, nhel, fac, mult, inta)
      integer, intent(inout) :: index, fac
      integer, intent(in) :: nhel
      integer, dimension(:), intent(in) :: mult
      integer, dimension(nhel,size(mult)), intent(out) :: inta
      integer :: i, j
      if (index > size(mult)) return
      fac = fac / mult(index)
      do j = 1, nhel
         inta(j,index) = sj (mult(index),mod(((j-1)/fac),mult(index))+1)
      end do
      index = index + 1
      call create_spin_table (index, nhel, fac, mult, inta)
    end subroutine create_spin_table
  end subroutine write_unit_matrix_element
```

13.1.8   Interface file for the generated modules

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_write_driver
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_write_driver (prc_lib)

      type(process_library_t), intent(inout) :: prc_lib
      type(string_t) :: filename, prefix
      type(string_t), dimension(:), allocatable :: prc_id, model, restrictions
      integer, dimension(:), allocatable :: n_par
      character(32), dimension(:), allocatable :: md5sum
      type(process_configuration_t), pointer :: current
      integer :: u, i, n_prc

      call msg_message ("Writing interface code for process library '" // &
           char (process_library_get_name (prc_lib)) // "'")
      prefix = prc_lib%basename // "_"

      n_prc = prc_lib%n_prc
      allocate (prc_id (n_prc), model (n_prc), restrictions (n_prc))
      allocate (n_par (n_prc), md5sum (n_prc))
      current => prc_lib%prc_first
      do i = 1, n_prc
         prc_id(i) = current%id
         model(i) = model_get_name (current%model)
         restrictions(i) = current%restrictions
         n_par(i) = model_get_n_parameters (current%model)
```

```
      md5sum(i) = current%md5sum
      current => current%next
   end do
   filename = prc_lib%basename // "_interface.f90"
   u = free_unit ()
   open (unit=u, file=char(prc_lib%basename // ".f90"), action="write")
   write (u, "(A)")  "! WHIZARD process interface"
   write (u, "(A)")  "!"
   write (u, "(A)")  "! Automatically generated file, do not edit"
   call write_get_n_processes_fun ()
   call write_get_process_id_fun ()
   call write_get_model_name_fun ()
   call write_get_restrictions_fun ()
   call write_get_md5sum_fun ()
   call write_string_to_array_fun ()
   call write_get_int_fun ("n_in",  "number_particles_in")
   call write_get_int_fun ("n_out", "number_particles_out")
   call write_get_int_fun ("n_flv", "number_flavor_states")
   call write_get_int_fun ("n_hel", "number_spin_states")
   call write_get_int_fun ("n_col", "number_color_flows")
   call write_get_int_fun ("n_cin", "number_color_indices")
   call write_get_int_fun ("n_cf",  "number_color_factors")
   call write_set_int_sub1 ("flv_state", "flavor_states")
   call write_set_int_sub1 ("hel_state", "spin_states")
   call write_set_int_sub2 ("col_state", "color_flows", "ghost_flag")
   call write_set_cf_tab_sub ()
   call write_init_get_fptr ()
   call write_final_get_fptr ()
   call write_update_alpha_s_get_fptr ()
   call write_reset_helicity_selection_get_fptr ()
   call write_new_event_get_fptr ()
   call write_is_allowed_get_fptr ()
   call write_get_amplitude_get_fptr ()
   close (u)

   prc_lib%status = max (prc_lib%status, STAT_CODE_GENERATED)

contains

   subroutine write_get_n_processes_fun ()
      write (u, "(A)")  ""
      write (u, "(A)")  "! Return the number of processes in this library"
      write (u, "(A)")  "function " // char (prefix) &
           // "get_n_processes () result (n) bind(C)"
      write (u, "(A)")  "  use iso_c_binding"
      write (u, "(A)")  "  integer(c_int) :: n"
      write (u, "(A,I0)")  "  n = ", n_prc
      write (u, "(A)")  "end function " // char (prefix) &
           // "get_n_processes"
   end subroutine write_get_n_processes_fun

   subroutine write_get_process_id_fun ()
      write (u, "(A)")  ""
      write (u, "(A)")  "! Return the process ID of process #i (as a C pointer to a character arra
```

```
      write (u, "(A)")  "subroutine " // char (prefix) &
           // "get_process_id (i, cptr, len) bind(C)"
      write (u, "(A)")  "  use iso_c_binding"
      write (u, "(A)")  "  integer(c_int), intent(in) :: i"
      write (u, "(A)")  "  type(c_ptr), intent(inout) :: cptr"
      write (u, "(A)")  "  integer(c_int), intent(out) :: len"
      write (u, "(A)")  "  character(kind=c_char), dimension(:), allocatable, target, save :: a"
      call write_string_to_array_interface ()
      write (u, "(A)")  "  select case (i)"
      write (u, "(A)")  "  case (0);  if (allocated (a))  deallocate (a)"
      do i = 1, n_prc
         write (u, "(A,I0,A)")  "  case (", i, ");  " &
              // "call " // char (prefix) &
              // "string_to_array ('" // char (prc_id(i)) // "', a)"
      end do
      write (u, "(A)")  "  end select"
      write (u, "(A)")  "  if (allocated (a)) then"
      write (u, "(A)")  "     cptr = c_loc (a)"
      write (u, "(A)")  "     len = size (a)"
      write (u, "(A)")  "  else"
      write (u, "(A)")  "     cptr = c_null_ptr"
      write (u, "(A)")  "     len = 0"
      write (u, "(A)")  "  end if"
      write (u, "(A)")  "end subroutine " // char (prefix) &
           // "get_process_id"
    end subroutine write_get_process_id_fun

    subroutine write_get_model_name_fun ()
      write (u, "(A)")  ""
      write (u, "(A)")  "! Return the model name for process #i (as a C pointer to a character arr
      write (u, "(A)")  "subroutine " // char (prefix) &
           // "get_model_name (i, cptr, len) bind(C)"
      write (u, "(A)")  "  use iso_c_binding"
      write (u, "(A)")  "  integer(c_int), intent(in) :: i"
      write (u, "(A)")  "  type(c_ptr), intent(inout) :: cptr"
      write (u, "(A)")  "  integer(c_int), intent(out) :: len"
      write (u, "(A)")  "  character(kind=c_char), dimension(:), allocatable, target, save :: a"
      call write_string_to_array_interface ()
      write (u, "(A)")  "  select case (i)"
      write (u, "(A)")  "  case (0);  if (allocated (a))  deallocate (a)"
      do i = 1, n_prc
         write (u, "(A,I0,A)")  "  case (", i, ");  " &
              // "call " // char (prefix) &
              // "string_to_array ('" // char (model(i)) // "', a)"
      end do
      write (u, "(A)")  "  end select"
      write (u, "(A)")  "  if (allocated (a)) then"
      write (u, "(A)")  "     cptr = c_loc (a)"
      write (u, "(A)")  "     len = size (a)"
      write (u, "(A)")  "  else"
      write (u, "(A)")  "     cptr = c_null_ptr"
      write (u, "(A)")  "     len = 0"
      write (u, "(A)")  "  end if"
      write (u, "(A)")  "end subroutine " // char (prefix) &
```

```
        // "get_model_name"
end subroutine write_get_model_name_fun

subroutine write_get_restrictions_fun ()
  write (u, "(A)")   ""
  write (u, "(A)")   "! Return the model name for process #i (as a C pointer to a character arr
  write (u, "(A)")   "subroutine " // char (prefix) &
       // "get_restrictions (i, cptr, len) bind(C)"
  write (u, "(A)")   "  use iso_c_binding"
  write (u, "(A)")   "  integer(c_int), intent(in) :: i"
  write (u, "(A)")   "  type(c_ptr), intent(inout) :: cptr"
  write (u, "(A)")   "  integer(c_int), intent(out) :: len"
  write (u, "(A)")   "  character(kind=c_char), dimension(:), allocatable, target, save :: a"
  call write_string_to_array_interface ()
  write (u, "(A)")   "  select case (i)"
  write (u, "(A)")   "  case (0);  if (allocated (a))  deallocate (a)"
  do i = 1, n_prc
     write (u, "(A,I0,A)")  "  case (", i, ");  " &
          // "call " // char (prefix) &
          // "string_to_array ('" // char (restrictions(i)) // "', a)"
  end do
  write (u, "(A)")   "  end select"
  write (u, "(A)")   "  if (allocated (a)) then"
  write (u, "(A)")   "     cptr = c_loc (a)"
  write (u, "(A)")   "     len = size (a)"
  write (u, "(A)")   "  else"
  write (u, "(A)")   "     cptr = c_null_ptr"
  write (u, "(A)")   "     len = 0"
  write (u, "(A)")   "  end if"
  write (u, "(A)")   "end subroutine " // char (prefix) &
       // "get_restrictions"
end subroutine write_get_restrictions_fun

subroutine write_get_md5sum_fun ()
  integer :: i
  write (u, "(A)")   ""
  write (u, "(A)")   "! Return the MD5 sum for the process configuration (as a C pointer to a c
  write (u, "(A)")   "subroutine " // char (prefix) &
       // "get_md5sum (i, cptr, len) bind(C)"
  write (u, "(A)")   "  use iso_c_binding"
  call write_use_lines ("md5sum", "md5sum")
  write (u, "(A)")   "  integer(c_int), intent(in) :: i"
  write (u, "(A)")   "  type(c_ptr), intent(inout) :: cptr"
  write (u, "(A)")   "  integer(c_int), intent(out) :: len"
  write (u, "(A)")   "  character(kind=c_char), dimension(:), allocatable, target, save :: a"
  call write_string_to_array_interface ()
  write (u, "(A)")   "  select case (i)"
  write (u, "(A)")   "  case (0);  if (allocated (a))  deallocate (a)"
  do i = 1, n_prc
     write (u, "(A,I0,A)")  "  case (", i, ");  " &
          // "call " // char (prefix) &
          // "string_to_array (" // char (prc_id(i)) &
          // "_md5sum (), a)"
  end do
```

```
    write (u, "(A)")  "  end select"
    write (u, "(A)")  "  if (allocated (a)) then"
    write (u, "(A)")  "     cptr = c_loc (a)"
    write (u, "(A)")  "     len = size (a)"
    write (u, "(A)")  "  else"
    write (u, "(A)")  "     cptr = c_null_ptr"
    write (u, "(A)")  "     len = 0"
    write (u, "(A)")  "  end if"
    write (u, "(A)")  "end subroutine " // char (prefix) &
         // "get_md5sum"
  end subroutine write_get_md5sum_fun

  subroutine write_string_to_array_interface ()
    write (u, "(2x,A)")  "interface"
    write (u, "(5x,A)")  "subroutine " // char (prefix) &
         // "string_to_array (string, a)"
    write (u, "(5x,A)")  "  use iso_c_binding"
    write (u, "(5x,A)")  "  character(*), intent(in) :: string"
    write (u, "(5x,A)")  "  character(kind=c_char), dimension(:), allocatable, intent(out) :: a"
    write (u, "(5x,A)")  "end subroutine " // char (prefix) &
         // "string_to_array"
    write (u, "(2x,A)")  "end interface"
  end subroutine write_string_to_array_interface

  subroutine write_string_to_array_fun ()
    write (u, "(A)")  ""
    write (u, "(A)")  "! Auxiliary: convert character string to array pointer"
    write (u, "(A)")  "subroutine " // char (prefix) &
         // "string_to_array (string, a)"
    write (u, "(A)")  "  use iso_c_binding"
    write (u, "(A)")  "  character(*), intent(in) :: string"
    write (u, "(A)")  "  character(kind=c_char), dimension(:), allocatable, intent(out) :: a"
    write (u, "(A)")  "  integer :: i"
    write (u, "(A)")  "  allocate (a (len (string)))"
    write (u, "(A)")  "  do i = 1, size (a)"
    write (u, "(A)")  "     a(i) = string(i:i)"
    write (u, "(A)")  "  end do"
    write (u, "(A)")  "end subroutine " // char (prefix) &
         // "string_to_array"
  end subroutine write_string_to_array_fun

  subroutine write_get_int_fun (vname, fname)
    character(*), intent(in) :: vname, fname
    write (u, "(A)")  ""
    write (u, "(A)")  "! Return the value of " // vname
    write (u, "(A)")  "function " // char (prefix) &
         // "get_" // vname // " (pid)" &
         // " result (" // vname // ") bind(C)"
    write (u, "(A)")  "  use iso_c_binding"
    call write_use_lines (vname, fname)
    write (u, "(A)")  "  integer(c_int), intent(in) :: pid"
    write (u, "(A)")  "  integer(c_int) :: " // vname
    call write_case_lines (vname // " = ", "_" // vname // " ()")
    write (u, "(A)")  "end function " // char (prefix) &
```

```
                // "get_" // vname
        end subroutine write_get_int_fun

        subroutine write_set_int_sub1 (vname, fname)
          character(*), intent(in) :: vname, fname
          write (u, "(A)")  ""
          write (u, "(A)")  "! Set table: " // vname
          write (u, "(A)")  "subroutine " // char (prefix) &
                // "set_" // vname &
                // " (pid, cptr, shape) bind(C)"
          write (u, "(A)")  "  use iso_c_binding"
          call write_use_lines (vname, fname)
          write (u, "(A)")  "  integer(c_int), intent(in) :: pid"
          write (u, "(A)")  "  type(c_ptr), intent(in) :: cptr"
          write (u, "(A)")  "  integer(c_int), dimension(2), intent(in) :: shape"
          write (u, "(A)")  "  integer(c_int), dimension(:,:), pointer :: " // vname
          if (kind(1) /= c_int) then
             write (u, "(A)")  "  integer, dimension(:,:), allocatable :: " &
                   // vname // "_tmp"
          end if
          write (u, "(A)")  "  call c_f_pointer (cptr, " // vname // ", shape)"
          if (kind(1) == c_int) then
             call write_case_lines ("call ", "_" // vname // " (" // vname // ")")
          else
             write (u, "(A)")  "  allocate (" &
                   // vname // "_tmp (shape(1), shape(2)))"
             call write_case_lines ("call ", &
                   "_" // vname // " (" // vname // "_tmp)")
             write (u, "(A)")  "  " // vname // " = " // vname // "_tmp"
          end if
          write (u, "(A)")  "end subroutine " // char (prefix) &
                // "set_" // vname
        end subroutine write_set_int_sub1

        subroutine write_set_int_sub2 (vname, fname, lname)
          character(*), intent(in) :: vname, fname, lname
          write (u, "(A)")  ""
          write (u, "(A)")  "! Set tables: " // vname // ", " // lname
          write (u, "(A)")  "subroutine " // char (prefix) &
                // "set_" // vname &
                // " (pid, cptr, shape, lcptr, lshape) bind(C)"
          write (u, "(A)")  "  use iso_c_binding"
          call write_use_lines (vname, fname)
          write (u, "(A)")  "  integer(c_int), intent(in) :: pid"
          write (u, "(A)")  "  type(c_ptr), intent(in) :: cptr"
          write (u, "(A)")  "  integer(c_int), dimension(3), intent(in) :: shape"
          write (u, "(A)")  "  type(c_ptr), intent(in) :: lcptr"
          write (u, "(A)")  "  integer(c_int), dimension(2), intent(in) :: lshape"
          write (u, "(A)")  "  integer(c_int), dimension(:,:,:), pointer :: " &
                // vname
          write (u, "(A)")  "  logical(c_bool), dimension(:,:), pointer :: " &
                // lname
          if (kind(1) /= c_int) then
             write (u, "(A)")  "  integer, dimension(:,:), allocatable :: " &
```

```
               // vname // "_tmp"
       end if
       if (kind(.true.) /= c_bool) then
          write (u, "(A)")  "  logical, dimension(:,:), allocatable :: " &
                // lname // "_tmp"
       end if
       write (u, "(A)")  "  call c_f_pointer (cptr, " // vname // ", shape)"
       write (u, "(A)")  "  call c_f_pointer (lcptr, " // lname // ", lshape)"
       if (kind(1) /= c_int) then
          write (u, "(A)")  "  allocate (" &
                // vname // "_tmp (shape(1), shape(2), shape(3)))"
       end if
       if (kind(.true.) /= c_bool) then
          write (u, "(A)")  "  allocate (" &
                // lname // "_tmp (lshape(1), lshape(2)))"
       end if
       if (kind(1) == c_int) then
          if (kind(.true.) == c_bool) then
             call write_case_lines ("call ", &
                   "_" // vname // " (" // vname // ", " // lname // ")")
          else
             call write_case_lines ("call ", &
                   "_" // vname // " (" // vname // ", " // lname // "_tmp)")
             write (u, "(A)")  "  " // lname // " = " // lname // "_tmp"
          end if
       else
          if (kind(.true.) == c_bool) then
             call write_case_lines ("call ", &
                   "_" // vname // " (" // vname // "_tmp, " // lname // ")")
          else
             call write_case_lines ("call ", &
                   "_" // vname // " (" // vname // "_tmp, " // lname // "_tmp)")
             write (u, "(A)")  "  " // lname // " = " // lname // "_tmp"
          end if
          write (u, "(A)")  "  " // vname // " = " // vname // "_tmp"
       end if
       write (u, "(A)")  "end subroutine " // char (prefix) &
             // "set_" // vname
   end subroutine write_set_int_sub2

   subroutine write_set_cf_tab_sub ()
       write (u, "(A)")  ""
       write (u, "(A)")  "subroutine " // char (prefix) &
             // "set_cf_table (pid, iptr1, iptr2, cptr, shape) bind(C)"
       write (u, "(A)")  "  use iso_c_binding"
       write (u, "(A)")  "  use kinds"
       write (u, "(A)")  "  use omega_color"
       call write_use_lines ("color_factors", "color_factors")
       write (u, "(A)")  "  integer(c_int), intent(in) :: pid"
       write (u, "(A)")  "  type(c_ptr), intent(in) :: iptr1, iptr2, cptr"
       write (u, "(A)")  "  integer(c_int), dimension(1), intent(in) :: shape"
       write (u, "(A)")  "  integer(c_int), dimension(:), pointer :: " &
             // "cf_index1, cf_index2"
       write (u, "(A)")  "  complex(c_default_complex), dimension(:), " &
```

```
      // "pointer :: col_factor"
  write (u, "(A)")  "  type(omega_color_factor), dimension(:), " &
      // "allocatable :: cf"
  write (u, "(A)")  "  call c_f_pointer (iptr1, cf_index1, shape)"
  write (u, "(A)")  "  call c_f_pointer (iptr2, cf_index2, shape)"
  write (u, "(A)")  "  call c_f_pointer (cptr, col_factor, shape)"
  write (u, "(A)")  "  allocate (cf (shape(1)))"
  call write_case_lines ("call ", "_color_factors (cf)")
  write (u, "(A)")  "  cf_index1 = cf%i1"
  write (u, "(A)")  "  cf_index2 = cf%i2"
  write (u, "(A)")  "  col_factor = cf%factor"
  write (u, "(A)")  "end subroutine " // char (prefix) // "set_cf_table"
end subroutine write_set_cf_tab_sub

subroutine write_init_get_fptr ()
  write (u, "(A)")  ""
  write (u, "(A)")  "! Return pointer to function: 'init'"
  write (u, "(A)")  "subroutine " // char (prefix) &
      // "init_get_fptr (pid, fptr) bind(C)"
  write (u, "(A)")  "  use iso_c_binding"
  write (u, "(A)")  "  integer(c_int), intent(in) :: pid"
  write (u, "(A)")  "  type(c_funptr), intent(out) :: fptr"
  write (u, "(A)")  "  abstract interface"
  write (u, "(A)")  "     subroutine prc_init (par) bind(C)"
  write (u, "(A)")  "       use iso_c_binding"
  write (u, "(A)")  "       use kinds"
  write (u, "(A)")  "       real(c_default_float), dimension(*), " &
      // "intent(in) :: par"
  write (u, "(A)")  "     end subroutine prc_init"
  write (u, "(A)")  "  end interface"
  do i = 1, n_prc
     write (u, "(2x,A)")  "procedure(prc_init), bind(C) :: " &
         // char (prc_id(i)) // "_init"
  end do
  call write_case_lines ("fptr = c_funloc (", "_init)")
  write (u, "(A)")  "end subroutine " // char (prefix) &
      // "init_get_fptr"
  do i = 1, n_prc
     write (u, *)
     write (u, "(A)")  "subroutine " // char (prc_id(i)) &
         // "_init (par) bind(C)"
     write (u, "(A)")  "  use iso_c_binding"
     write (u, "(A)")  "  use kinds"
     write (u, "(A)")  "  use " // char (prc_id(i))
     write (u, "(A)")  "  real(c_default_float), dimension(*), " &
         // "intent(in) :: par"
     if (c_default_float == default) then
        write (u, "(A)")  "  call init (par)"
     else
        write (u, "(A, I0)")  "  integer, parameter :: n_par = ", n_par(i)
        write (u, "(A)")  "  real(default), dimension(n_par) :: fpar"
        write (u, "(A)")  "  fpar = par"
        write (u, "(A)")  "  call init (fpar)"
     end if
```

```
      write (u, "(A)")  "end subroutine " // char (prc_id(i)) // "_init"
    end do
  end subroutine write_init_get_fptr

  subroutine write_final_get_fptr ()
    write (u, "(A)")  ""
    write (u, "(A)")  "! Return pointer to function: 'final'"
    write (u, "(A)")  "subroutine " // char (prefix) &
         // "final_get_fptr (pid, fptr) bind(C)"
    write (u, "(A)")  "  use iso_c_binding"
    write (u, "(A)")  "  integer(c_int), intent(in) :: pid"
    write (u, "(A)")  "  type(c_funptr), intent(out) :: fptr"
    write (u, "(A)")  "  abstract interface"
    write (u, "(A)")  "     subroutine prc_final () bind(C)"
    write (u, "(A)")  "     end subroutine prc_final"
    write (u, "(A)")  "  end interface"
    do i = 1, n_prc
       write (u, "(2x,A)")  "procedure(prc_final), bind(C) :: " &
            // char (prc_id(i)) // "_final"
    end do
    call write_case_lines ("fptr = c_funloc (", "_final)")
    write (u, "(A)")  "end subroutine " // char (prefix) &
         // "final_get_fptr"
    do i = 1, n_prc
       write (u, *)
       write (u, "(A)")  "subroutine " // char (prc_id(i)) &
            // "_final () bind(C)"
       write (u, "(A)")  "  use " // char (prc_id(i))
       write (u, "(A)")  "  call final ()"
       write (u, "(A)")  "end subroutine " // char (prc_id(i)) // "_final"
    end do
  end subroutine write_final_get_fptr

  subroutine write_update_alpha_s_get_fptr ()
    write (u, "(A)")  ""
    write (u, "(A)")  "! Return pointer to function: 'update_alpha_s'"
    write (u, "(A)")  "subroutine " // char (prefix) &
         // "update_alpha_s_get_fptr (pid, fptr) bind(C)"
    write (u, "(A)")  "  use iso_c_binding"
    write (u, "(A)")  "  integer(c_int), intent(in) :: pid"
    write (u, "(A)")  "  type(c_funptr), intent(out) :: fptr"
    write (u, "(A)")  "  abstract interface"
    write (u, "(A)")  "     subroutine prc_update_alpha_s (alpha_s) bind(C)"
    write (u, "(A)")  "       use iso_c_binding"
    write (u, "(A)")  "       use kinds"
    write (u, "(A)")  "       real(c_default_float), " &
         // "intent(in) :: alpha_s"
    write (u, "(A)")  "     end subroutine prc_update_alpha_s"
    write (u, "(A)")  "  end interface"
    do i = 1, n_prc
       write (u, "(2x,A)")  "procedure(prc_update_alpha_s), bind(C) :: " &
            // char (prc_id(i)) // "_update_alpha_s"
    end do
    call write_case_lines ("fptr = c_funloc (", "_update_alpha_s)")
```

```
      write (u, "(A)")  "end subroutine " // char (prefix) &
           // "update_alpha_s_get_fptr"
      do i = 1, n_prc
         write (u, *)
         write (u, "(A)")  "subroutine " // char (prc_id(i)) &
              // "_update_alpha_s (alpha_s) bind(C)"
         write (u, "(A)")  "  use iso_c_binding"
         write (u, "(A)")  "  use kinds"
         write (u, "(A)")  "  use " // char (prc_id(i))
         write (u, "(A)")  "  real(c_default_float), " &
              // "intent(in) :: alpha_s"
         if (c_default_float == default) then
           write (u, "(A)")  "  call update_alpha_s (alpha_s)"
         else
           write (u, "(A)")  "  call update_alpha_s " &
                // "(real (alpha_s, c_default_float))"
         end if
         write (u, "(A)")  "end subroutine " // char (prc_id(i)) &
              // "_update_alpha_s"
      end do
    end subroutine write_update_alpha_s_get_fptr

    subroutine write_reset_helicity_selection_get_fptr ()
      write (u, "(A)")  ""
      write (u, "(A)")  "! Return pointer to function: " &
           // "'reset_helicity_selection'"
      write (u, "(A)")  "subroutine " // char (prefix) &
           // "reset_helicity_selection_get_fptr (pid, fptr) bind(C)"
      write (u, "(A)")  "  use iso_c_binding"
      write (u, "(A)")  "  integer(c_int), intent(in) :: pid"
      write (u, "(A)")  "  type(c_funptr), intent(out) :: fptr"
      write (u, "(A)")  "  abstract interface"
      write (u, "(A)")  "     subroutine " &
           // "prc_reset_helicity_selection (threshold, cutoff) bind(C)"
      write (u, "(A)")  "       use iso_c_binding"
      write (u, "(A)")  "       use kinds"
      write (u, "(A)")  "       real(c_default_float), " &
           // "intent(in) :: threshold"
      write (u, "(A)")  "       integer(c_int), " &
           // "intent(in) :: cutoff"
      write (u, "(A)")  "     end subroutine prc_reset_helicity_selection"
      write (u, "(A)")  "  end interface"
      do i = 1, n_prc
         write (u, "(2x,A)")  "procedure(prc_reset_helicity_selection), " &
              // "bind(C) :: " &
              // char (prc_id(i)) // "_reset_helicity_selection"
      end do
      call write_case_lines ("fptr = c_funloc (", "_reset_helicity_selection)")
      write (u, "(A)")  "end subroutine " // char (prefix) &
           // "reset_helicity_selection_get_fptr"
      do i = 1, n_prc
         write (u, *)
         write (u, "(A)")  "subroutine " // char (prc_id(i)) &
              // "_reset_helicity_selection (threshold, cutoff) bind(C)"
```

```
      write (u, "(A)")  "  use iso_c_binding"
      write (u, "(A)")  "  use kinds"
      write (u, "(A)")  "  use " // char (prc_id(i))
      write (u, "(A)")  "  real(c_default_float), " &
           // "intent(in) :: threshold"
      write (u, "(A)")  "  integer(c_int), " &
           // "intent(in) :: cutoff"
      write (u, "(A)")  "  real(default) :: rthreshold"
      write (u, "(A)")  "  integer :: icutoff"
      write (u, "(A)")  "  rthreshold = threshold"
      write (u, "(A)")  "  icutoff = cutoff"
      write (u, "(A)")  "  call reset_helicity_selection " &
           // "(rthreshold, icutoff)"
      write (u, "(A)")  "end subroutine " // char (prc_id(i)) &
           // "_reset_helicity_selection"
    end do
  end subroutine write_reset_helicity_selection_get_fptr

  subroutine write_new_event_get_fptr ()
    write (u, "(A)")  ""
    write (u, "(A)")  "! Return pointer to function: 'new_event'"
    write (u, "(A)")  "subroutine " // char (prefix) &
         // "new_event_get_fptr (pid, fptr) bind(C)"
    write (u, "(A)")  "  use iso_c_binding"
    write (u, "(A)")  "  integer(c_int), intent(in) :: pid"
    write (u, "(A)")  "  type(c_funptr), intent(out) :: fptr"
    write (u, "(A)")  "  abstract interface"
    write (u, "(A)")  "     subroutine prc_new_event (p) bind(C)"
    write (u, "(A)")  "       use iso_c_binding"
    write (u, "(A)")  "       use kinds"
    write (u, "(A)")  "       real(c_default_float), dimension(0:3,*), " &
         // "intent(in) :: p"
    write (u, "(A)")  "     end subroutine prc_new_event"
    write (u, "(A)")  "  end interface"
    do i = 1, n_prc
       write (u, "(2x,A)")  "procedure(prc_new_event), bind(C) :: " &
            // char (prc_id(i)) // "_new_event"
    end do
    call write_case_lines ("fptr = c_funloc (", "_new_event)")
    write (u, "(A)")  "end subroutine " // char (prefix) &
         // "new_event_get_fptr"
    do i = 1, n_prc
       write (u, *)
       write (u, "(A)")  "subroutine " // char (prc_id(i)) &
            // "_new_event (p) bind(C)"
       write (u, "(A)")  "  use iso_c_binding"
       write (u, "(A)")  "  use kinds"
       write (u, "(A)")  "  use " // char (prc_id(i))
       write (u, "(A)")  "  real(c_default_float), dimension(0:3,*), " &
            // "intent(in) :: p"
       if (c_default_float == default) then
          write (u, "(A)")  "  call new_event (p)"
       else
          write (u, "(A)")  "  integer :: n_tot"
```

```
              write (u, "(A)")  "  real(default), dimension(:,:), " &
                   // "allocatable :: k"
              write (u, "(A)")  "  n_tot = " &
                   // "number_particles_in () + number_particles_out ()"
              write (u, "(A)")  "  allocate (k (0:3,n_tot))"
              write (u, "(A)")  "  k = p"
              write (u, "(A)")  "  call new_event (k)"
           end if
           write (u, "(A)")  "end subroutine " // char (prc_id(i)) // "_new_event"
        end do
    end subroutine write_new_event_get_fptr

    subroutine write_is_allowed_get_fptr ()
      write (u, "(A)")  ""
      write (u, "(A)")  "! Return pointer to function: 'is_allowed'"
      write (u, "(A)")  "subroutine " // char (prefix) &
           // "is_allowed_get_fptr (pid, fptr) bind(C)"
      write (u, "(A)")  "  use iso_c_binding"
      write (u, "(A)")  "  integer(c_int), intent(in) :: pid"
      write (u, "(A)")  "  type(c_funptr), intent(out) :: fptr"
      write (u, "(A)")  "  abstract interface"
      write (u, "(A)")  "     function " &
           // "prc_is_allowed (flv, hel, col) result (flag) bind(C)"
      write (u, "(A)")  "       use iso_c_binding"
      write (u, "(A)")  "       use kinds"
      write (u, "(A)")  "       logical(c_bool) :: flag"
      write (u, "(A)")  "       integer(c_int), intent(in) :: flv, hel, col"
      write (u, "(A)")  "     end function prc_is_allowed"
      write (u, "(A)")  "  end interface"
      do i = 1, n_prc
         write (u, "(2x,A)")  "procedure(prc_is_allowed), bind(C) :: " &
              // char (prc_id(i)) // "_is_allowed"
      end do
      call write_case_lines ("fptr = c_funloc (", "_is_allowed)")
      write (u, "(A)")  "end subroutine " // char (prefix) &
           // "is_allowed_get_fptr"
      do i = 1, n_prc
         write (u, *)
         write (u, "(A)")  "function " // char (prc_id(i)) &
              // "_is_allowed (flv, hel, col) result (flag) bind(C)"
         write (u, "(A)")  "  use iso_c_binding"
         write (u, "(A)")  "  use kinds"
         write (u, "(A)")  "  use " // char (prc_id(i))
         write (u, "(A)")  "  logical(c_bool) :: flag"
         write (u, "(A)")  "  integer(c_int), intent(in) :: flv, hel, col"
         if (c_int == kind(1)) then
            write (u, "(A)")  "  flag = is_allowed (flv, hel, col)"
         else
            write (u, "(A)")  "  integer :: iflv, ihel, icol"
            write (u, "(A)")  "  iflv = flv;  ihel = hel;  icol = col"
            write (u, "(A)")  "  flag = is_allowed (iflv, ihel, icol)"
         end if
         write (u, "(A)")  "end function " // char (prc_id(i)) &
              // "_is_allowed"
```

```
        end do
  end subroutine write_is_allowed_get_fptr

  subroutine write_get_amplitude_get_fptr ()
    write (u, "(A)")  ""
    write (u, "(A)")  "! Return pointer to function: 'get_amplitude'"
    write (u, "(A)")  "subroutine " // char (prefix) &
         // "get_amplitude_get_fptr (pid, fptr) " &
         // "bind(C)"
    write (u, "(A)")  "  use iso_c_binding"
    write (u, "(A)")  "  integer(c_int), intent(in) :: pid"
    write (u, "(A)")  "  type(c_funptr), intent(out) :: fptr"
    write (u, "(A)")  "  abstract interface"
    write (u, "(A)")  "     function " &
         // "prc_get_amplitude (flv, hel, col) result (amp) bind(C)"
    write (u, "(A)")  "       use iso_c_binding"
    write (u, "(A)")  "       use kinds"
    write (u, "(A)")  "       complex(c_default_complex) :: amp"
    write (u, "(A)")  "       integer(c_int), intent(in) :: flv, hel, col"
    write (u, "(A)")  "     end function prc_get_amplitude"
    write (u, "(A)")  "  end interface"
    do i = 1, n_prc
       write (u, "(2x,A)") "procedure(prc_get_amplitude), bind(C) :: " &
            // char (prc_id(i)) // "_get_amplitude"
    end do
    call write_case_lines ("fptr = c_funloc (", "_get_amplitude)")
    write (u, "(A)")  "end subroutine " // char (prefix) &
         // "get_amplitude_get_fptr"
    do i = 1, n_prc
       write (u, *)
       write (u, "(A)")  "function " // char (prc_id(i)) &
            // "_get_amplitude (flv, hel, col) result (amp) bind(C)"
       write (u, "(A)")  "  use iso_c_binding"
       write (u, "(A)")  "  use kinds"
       write (u, "(A)")  "  use " // char (prc_id(i))
       write (u, "(A)")  "  complex(c_default_complex) :: amp"
       write (u, "(A)")  "  integer(c_int), intent(in) :: flv, hel, col"
       if (c_int == kind(1)) then
          write (u, "(A)")  "  amp = get_amplitude (flv, hel, col)"
       else
          write (u, "(A)")  "  integer :: iflv, ihel, icol"
          write (u, "(A)")  "  iflv = flv;  ihel = hel;  icol = col"
          write (u, "(A)")  "  amp = get_amplitude (iflv, ihel, icol)"
       end if
       write (u, "(A)")  "end function " // char (prc_id(i)) &
            // "_get_amplitude"
    end do
  end subroutine write_get_amplitude_get_fptr

  subroutine write_use_lines (vname, fname)
    character(*), intent(in) :: vname, fname
    integer :: i
    do i = 1, n_prc
       write (u, "(2x,A)") "use " // char (prc_id(i)) // ", only: " &
```

974

```
               // char (prc_id(i)) // "_" // vname // " => " // fname
         end do
      end subroutine write_use_lines

      subroutine write_case_lines (cmd1, cmd2)
        character(*), intent(in) :: cmd1, cmd2
        integer :: i
        write (u, "(A)")  "  select case (pid)"
        do i = 1, n_prc
           write (u, "(2x,A,I0,A)")  "case(", i, ");  " &
                // cmd1 // char (prc_id(i)) // cmd2
        end do
        write (u, "(A)")  "  end select"
      end subroutine write_case_lines

   end subroutine process_library_write_driver
```

13.1.9 Library manager

When static libraries are compiled, procedure pointer are not assigned by a
dlopen mechanism, but must be done at program startup. Mainly for this task
we write a library manager which links to the static libraries as they are defined
by the user.

For each library, it has to assign all possible interface function to a C function
pointer, which then is dereferenced in the same way as it is done for dlopened
libraries.

⟨*Process libraries: public*⟩+≡
```
    public :: write_library_manager
```

⟨*Process libraries: procedures*⟩+≡
```
   subroutine write_library_manager (libname)

     type(string_t), dimension(:), intent(in) :: libname
     integer :: u, i

     call msg_message ("Writing library manager code")
     u = free_unit ()
     open (unit=u, file="libmanager.f90", action="write", status="replace")
     write (u, "(A)")  "! WHIZARD library manager"
     write (u, "(A)")  "!"
     write (u, "(A)")  "! Automatically generated file, do not edit"
     write (u, "(A)")  ""
     write (u, "(A)")  "function libmanager_get_n_libs () result (n)"
     write (u, "(A)")  "  integer :: n"
     write (u, "(A,1x,I0)")  "  n =", size (libname)
     write (u, "(A)")  "end function libmanager_get_n_libs"
     write (u, "(A)")  ""
     write (u, "(A)")  "function libmanager_get_libname (i) result (name)"
     write (u, "(A)")  "  use iso_varying_string, string_t => varying_string"
     write (u, "(A)")  "  type(string_t) :: name"
     write (u, "(A)")  "  integer, intent(in) :: i"
     write (u, "(A)")  "  select case (i)"
```

```
      do i = 1, size (libname)
         call write_lib_name (i, libname(i))
      end do
      write (u, "(A)")  "  case default;  name = ''"
      write (u, "(A)")  "  end select"
      write (u, "(A)")  "end function libmanager_get_libname"
      write (u, "(A)")  ""
      write (u, "(A)")  "function libmanager_get_c_funptr (libname, fname) " & 
           // "result (c_fptr)"
      write (u, "(A)")  "  use iso_c_binding"
      write (u, "(A)")  "  use prclib_interfaces"
      write (u, "(A)")  "  type(c_funptr) :: c_fptr"
      write (u, "(A)")  "  character(*), intent(in) :: libname, fname"
      do i = 1, size (libname)
         call write_lib_declarations (libname(i))
      end do
      write (u, "(A)")  "  select case (libname)"
      do i = 1, size (libname)
         call write_lib_code (libname(i))
      end do
      write (u, "(A)")  "  case default"
      write (u, "(A)")  "     c_fptr = c_null_funptr"
      write (u, "(A)")  "  end select"
      write (u, "(A)")  "end function libmanager_get_c_funptr"
      close (u)

    contains

      subroutine write_lib_name (i, libname)
        integer, intent(in) :: i
        type(string_t), intent(in) :: libname
        write (u, "(A,I0,A)")  "  case (", i, ");  name = '" // char (libname) & 
             // "'"
      end subroutine write_lib_name

      subroutine write_lib_declarations (libname)
        type(string_t), intent(in) :: libname
        write (u, "(A)")  "  procedure(prc_get_n_processes), bind(C) :: " & 
             // char (libname)// "_" //  "get_n_processes"
        write (u, "(A)")  "  procedure(prc_get_stringptr), bind(C) :: " & 
             // char (libname)// "_" //  "get_process_id"
        write (u, "(A)")  "  procedure(prc_get_stringptr), bind(C) :: " & 
             // char (libname)// "_" //  "get_model_name"
        write (u, "(A)")  "  procedure(prc_get_stringptr), bind(C) :: " & 
             // char (libname)// "_" //  "get_restrictions"
        write (u, "(A)")  "  procedure(prc_get_stringptr), bind(C) :: " & 
             // char (libname)// "_" //  "get_md5sum"
        write (u, "(A)")  "  procedure(prc_get_int), bind(C) :: " & 
             // char (libname)// "_" //  "get_n_in"
        write (u, "(A)")  "  procedure(prc_get_int), bind(C) :: " & 
             // char (libname)// "_" //  "get_n_out"
        write (u, "(A)")  "  procedure(prc_get_int), bind(C) :: " & 
             // char (libname)// "_" //  "get_n_flv"
        write (u, "(A)")  "  procedure(prc_get_int), bind(C) :: " & 
```

```
            // char (libname)// "_" //  "get_n_hel"
    write (u, "(A)")  "  procedure(prc_get_int), bind(C) :: " &
            // char (libname)// "_" //  "get_n_col"
    write (u, "(A)")  "  procedure(prc_get_int), bind(C) :: " &
            // char (libname)// "_" //  "get_n_cin"
    write (u, "(A)")  "  procedure(prc_get_int), bind(C) :: " &
            // char (libname)// "_" //  "get_n_cf"
    write (u, "(A)")  "  procedure(prc_set_int_tab1), bind(C) :: " &
            // char (libname)// "_" //  "set_flv_state"
    write (u, "(A)")  "  procedure(prc_set_int_tab1), bind(C) :: " &
            // char (libname)// "_" //  "set_hel_state"
    write (u, "(A)")  "  procedure(prc_set_int_tab2), bind(C) :: " &
            // char (libname)// "_" //  "set_col_state"
    write (u, "(A)")  "  procedure(prc_set_cf_tab), bind(C) :: " &
            // char (libname)// "_" //  "set_cf_table"
    write (u, "(A)")  "  procedure(prc_get_fptr), bind(C) :: " &
            // char (libname)// "_" //  "init_get_fptr"
    write (u, "(A)")  "  procedure(prc_get_fptr), bind(C) :: " &
            // char (libname)// "_" //  "final_get_fptr"
    write (u, "(A)")  "  procedure(prc_get_fptr), bind(C) :: " &
            // char (libname)// "_" //  "update_alpha_s_get_fptr"
    write (u, "(A)")  "  procedure(prc_get_fptr), bind(C) :: " &
            // char (libname)// "_" //  "new_event_get_fptr"
    write (u, "(A)")  "  procedure(prc_get_fptr), bind(C) :: " &
            // char (libname)// "_" //  "reset_helicity_selection_get_fptr"
    write (u, "(A)")  "  procedure(prc_get_fptr), bind(C) :: " &
            // char (libname)// "_" //  "is_allowed_get_fptr"
    write (u, "(A)")  "  procedure(prc_get_fptr), bind(C) :: " &
            // char (libname)// "_" //  "get_amplitude_get_fptr"
  end subroutine write_lib_declarations

  subroutine write_lib_code (libname)
    type(string_t), intent(in) :: libname
    write (u, "(2x,A)")  "case ('" // char (libname) // "')"
    write (u, "(2x,A)")  "   select case (fname)"
    call write_fun_code (char (libname), "get_n_processes")
    call write_fun_code (char (libname), "get_process_id")
    call write_fun_code (char (libname), "get_model_name")
    call write_fun_code (char (libname), "get_restrictions")
    call write_fun_code (char (libname), "get_md5sum")
    call write_fun_code (char (libname), "get_n_in")
    call write_fun_code (char (libname), "get_n_out")
    call write_fun_code (char (libname), "get_n_flv")
    call write_fun_code (char (libname), "get_n_hel")
    call write_fun_code (char (libname), "get_n_col")
    call write_fun_code (char (libname), "get_n_cin")
    call write_fun_code (char (libname), "get_n_cf")
    call write_fun_code (char (libname), "set_flv_state")
    call write_fun_code (char (libname), "set_hel_state")
    call write_fun_code (char (libname), "set_col_state")
    call write_fun_code (char (libname), "set_cf_table")
    call write_fun_code (char (libname), "init_get_fptr")
    call write_fun_code (char (libname), "final_get_fptr")
    call write_fun_code (char (libname), "update_alpha_s_get_fptr")
```

```
      call write_fun_code (char (libname), "reset_helicity_selection_get_fptr")
      call write_fun_code (char (libname), "new_event_get_fptr")
      call write_fun_code (char (libname), "is_allowed_get_fptr")
      call write_fun_code (char (libname), "get_amplitude_get_fptr")
      write (u, "(2x,A)")  "  case default"
      write (u, "(2x,A)")  "     print *, fname"
      write (u, "(2x,A)")  "     stop 'WHIZARD bug: " &
           // "libmanager cannot handle this function'"
      write (u, "(2x,A)")  "  end select"
    end subroutine write_lib_code

    subroutine write_fun_code (prefix, fname)
      character(*), intent(in) :: prefix, fname
      write (u, "(5x,A)")  "case ('" // fname // "')"
      write (u, "(5x,A)")  "  c_fptr = c_funloc (" // prefix &
           // "_" // fname // ")"
    end subroutine write_fun_code

  end subroutine write_library_manager
```

These are the interfaces of the functions provided by the library manager.

⟨*Process libraries: interfaces*⟩≡

```
    interface
      function libmanager_get_n_libs () result (n)
        integer :: n
      end function libmanager_get_n_libs
    end interface
```

⟨*Process libraries: interfaces*⟩+≡

```
    interface
      function libmanager_get_libname (i) result (name)
        use iso_varying_string, string_t => varying_string !NODEP!
        type(string_t) :: name
        integer, intent(in) :: i
      end function libmanager_get_libname
    end interface
```

⟨*Process libraries: interfaces*⟩+≡

```
    interface
      function libmanager_get_c_funptr (libname, fname) result (c_fptr)
        use iso_c_binding !NODEP!
        type(c_funptr) :: c_fptr
        character(*), intent(in) :: libname, fname
      end function libmanager_get_c_funptr
    end interface
```

13.1.10   Collect model-specific libraries

⟨*Process libraries: public*⟩+≡

```
    public :: get_modellibs_flags
```

⟨*Process libraries: procedures*⟩+≡

```
function get_modellibs_flags (prc_lib, os_data) result (flags)
  type(process_library_t), intent(in) :: prc_lib
  type(os_data_t), intent(in) :: os_data
  type(string_t) :: flags
  type(string_t), dimension(:), allocatable :: models
  type(string_t) :: modelname, modellib, modellib_full
  logical :: exist
  type(process_configuration_t), pointer :: current
  integer :: i, j, mi
  flags = ""
  if ((.not. os_data%use_testfiles) .and. &
            os_dir_exist (os_data%whizard_models_libpath_local)) &
              flags = flags // " -L" // os_data%whizard_models_libpath_local
  flags = flags // " -L" // os_data%whizard_models_libpath
  allocate (models(prc_lib%n_prc + 1))
  models = ""
  mi = 1
  current => prc_lib%prc_first
  SCAN: do i = 1, prc_lib%n_prc
    modelname = model_get_name (current%model)
    do j = 1, mi
      if (models(mi) == modelname) cycle SCAN
    end do
    models(mi) = modelname
    mi = mi + 1
    if (os_data%use_libtool) then
      modellib = "libparameters_" // modelname // ".la"
    else
      modellib = "libparameters_" // modelname // ".a"
    end if
    exist = .false.
    if (.not. os_data%use_testfiles) then
      modellib_full = os_data%whizard_models_libpath_local &
        // "/" // modellib
      inquire (file=char (modellib_full), exist=exist)
    end if
    if (.not. exist) then
      modellib_full = os_data%whizard_models_libpath &
        // "/" // modellib
      inquire (file=char (modellib_full), exist=exist)
    end if
    if (exist) flags = flags // " -lparameters_" // modelname
    current => current%next
  end do SCAN
  deallocate (models)
end function get_modellibs_flags
```

13.1.11 Compile and link a library

The process library proper consists of the process-specific Fortran source files
and the driver (interface)

⟨*Process libraries: public*⟩+≡

```
        public :: process_library_compile
⟨Process libraries: procedures⟩+≡
        subroutine process_library_compile &
            (prc_lib, os_data, recompile_library, objlist_link)
          type(process_library_t), intent(inout) :: prc_lib
          type(os_data_t), intent(in) :: os_data
          logical, intent(in) :: recompile_library
          type(string_t), intent(out) :: objlist_link
          type(string_t) :: objlist_comp
          type(process_configuration_t), pointer :: current
          type(string_t) :: ext
          integer :: i
          if (prc_lib%status == STAT_LOADED)  call process_library_unload (prc_lib)
          call msg_message ("Compiling process library '" // &
              char (process_library_get_name (prc_lib)) // "'")
          objlist_comp = ""
          objlist_link = ""
          if (os_data%use_libtool) then
              ext = ".lo"
          else
              ext = os_data%obj_ext
          end if
          current => prc_lib%prc_first
          SCAN_PROCESSES: do i = 1, prc_lib%n_prc
              objlist_link = objlist_link // " " // current%id // ext
              if (recompile_library) &
                  current%status = min (STAT_CODE_GENERATED, current%status)
              if (current%status == STAT_CODE_GENERATED) then
                  objlist_comp = objlist_comp // " " // current%id // ext
                  call os_compile_shared (current%id, os_data)
                  current%status = STAT_COMPILED
              else
                  call msg_message ("Skipping process '" // char (current%id) &
                      // "' (object code exists)")
              end if
              current => current%next
          end do SCAN_PROCESSES
          if (objlist_comp /= "") then
              call os_compile_shared (prc_lib%basename, os_data)
              objlist_link = objlist_link // " " // prc_lib%basename // ext
          else
              call msg_message ("Skipping library '" &
                  // char (prc_lib%basename) &
                  // "' (no processes have been recompiled)")
              objlist_link = ""
          end if
          prc_lib%status = STAT_COMPILED
        end subroutine process_library_compile

⟨Process libraries: public⟩+≡
        public :: process_library_link
⟨Process libraries: procedures⟩+≡
        subroutine process_library_link (prc_lib, os_data, objlist)
```

```
      type(process_library_t), intent(in) :: prc_lib
      type(os_data_t), intent(in) :: os_data
      type(string_t), intent(in) :: objlist
      if (objlist /= "") then
         call os_link_shared (objlist // " " // &
              os_data%whizard_ldflags // " " // os_data%ldflags // &
              get_modellibs_flags (prc_lib, os_data),  prc_lib%basename, os_data)
      end if
   end subroutine process_library_link
```

13.1.12 Standalone executable

Compile the library bundle and link with the libraries as a standalone executable

⟨*Process libraries: public*⟩+≡
```
   public :: compile_library_manager
```

⟨*Process libraries: procedures*⟩+≡
```
   subroutine compile_library_manager (os_data)
     type(os_data_t), intent(in) :: os_data
     call msg_message ("Compiling library manager")
     call os_compile_shared (var_str ("libmanager"), os_data)
   end subroutine compile_library_manager
```

⟨*Process libraries: public*⟩+≡
```
   public :: link_executable
```

⟨*Process libraries: procedures*⟩+≡
```
   subroutine link_executable (libname, exec_name, flags, os_data)
     type(string_t), dimension(:), intent(in) :: libname
     type(string_t), intent(in) :: exec_name, flags
     type(os_data_t), intent(in) :: os_data
     type(string_t) :: objlist, ext_o, ext_a
     integer :: i
     if (os_data%use_libtool) then
        ext_o = ".lo"
        ext_a = ".la"
     else
        ext_o = ".o"
        ext_a = ".a"
     end if
     objlist = "libmanager" // ext_o
     do i = 1, size (libname)
        objlist = objlist // " " // libname(i) // ext_a
     end do
     print *, char (flags)
     call os_link_static (objlist // flags, exec_name, os_data)
   end subroutine link_executable
```

13.1.13 Loading a library

This loads a process library. We assume that it resides in the current directory.

Loading the library assigns all procedure pointers to procedures within the library.

Unloading is done by the finalizer.

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_load
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_load (prc_lib, os_data, model, var_list, ignore)
      type(process_library_t), intent(inout), target :: prc_lib
      type(os_data_t), intent(in) :: os_data
      type(model_t), pointer, optional :: model
      type(var_list_t), intent(inout), optional :: var_list
      logical, intent(in), optional :: ignore
      type(c_funptr) :: c_fptr
      type(model_t), pointer :: mdl
      type(string_t) :: prefix
      logical :: ignore_error
      ignore_error = .false.;  if (present (ignore))  ignore_error = ignore
      if (prc_lib%status == STAT_LOADED) then
         if (.not. ignore_error) then
            call msg_message ("Process library '" // char (prc_lib%basename) &
                 // "' is already loaded")
         end if
         return
      end if
      if (prc_lib%static) then
         call msg_message ("Loading static process library '" &
              // char (prc_lib%basename) // "'")
      else
         call msg_message ("Loading process library '" &
              // char (prc_lib%basename) // "'")
         prc_lib%libname = os_get_dlname (prc_lib%basename, os_data, ignore)
         if (prc_lib%libname == "")  return
         call dlaccess_init (prc_lib%dlaccess, var_str ("."), &
              prc_lib%libname, os_data)
         call process_library_check_dlerror (prc_lib)
      end if
      prefix = prc_lib%basename
      c_fptr = process_library_get_c_funptr &
           (prc_lib, prefix, var_str ("get_n_processes"))
      call c_f_procpointer (c_fptr, prc_lib%get_n_prc)
      c_fptr = process_library_get_c_funptr &
           (prc_lib, prefix, var_str ("get_process_id"))
      call c_f_procpointer (c_fptr, prc_lib%get_process_id)
      c_fptr = process_library_get_c_funptr &
           (prc_lib, prefix, var_str ("get_model_name"))
      call c_f_procpointer (c_fptr, prc_lib%get_model_name)
      c_fptr = process_library_get_c_funptr &
           (prc_lib, prefix, var_str ("get_restrictions"))
      call c_f_procpointer (c_fptr, prc_lib%get_restrictions)
      c_fptr = process_library_get_c_funptr &
           (prc_lib, prefix, var_str ("get_md5sum"))
      call c_f_procpointer (c_fptr, prc_lib%get_md5sum)
      c_fptr = process_library_get_c_funptr &
```

```
      (prc_lib, prefix, var_str ("get_n_in"))
call c_f_procpointer (c_fptr, prc_lib%get_n_in)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("get_n_out"))
call c_f_procpointer (c_fptr, prc_lib%get_n_out)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("get_n_flv"))
call c_f_procpointer (c_fptr, prc_lib%get_n_flv)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("get_n_hel"))
call c_f_procpointer (c_fptr, prc_lib%get_n_hel)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("get_n_col"))
call c_f_procpointer (c_fptr, prc_lib%get_n_col)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("get_n_cin"))
call c_f_procpointer (c_fptr, prc_lib%get_n_cin)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("get_n_cf"))
call c_f_procpointer (c_fptr, prc_lib%get_n_cf)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("set_flv_state"))
call c_f_procpointer (c_fptr, prc_lib%set_flv_state)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("set_hel_state"))
call c_f_procpointer (c_fptr, prc_lib%set_hel_state)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("set_col_state"))
call c_f_procpointer (c_fptr, prc_lib%set_col_state)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("set_cf_table"))
call c_f_procpointer (c_fptr, prc_lib%set_cf_table)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("init_get_fptr"))
call c_f_procpointer (c_fptr, prc_lib%init_get_fptr)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("final_get_fptr"))
call c_f_procpointer (c_fptr, prc_lib%final_get_fptr)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("update_alpha_s_get_fptr"))
call c_f_procpointer (c_fptr, prc_lib%update_alpha_s_get_fptr)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("new_event_get_fptr"))
call c_f_procpointer (c_fptr, prc_lib%new_event_get_fptr)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("reset_helicity_selection_get_fptr"))
call c_f_procpointer (c_fptr, prc_lib%reset_helicity_selection_get_fptr)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("is_allowed_get_fptr"))
call c_f_procpointer (c_fptr, prc_lib%is_allowed_get_fptr)
c_fptr = process_library_get_c_funptr &
      (prc_lib, prefix, var_str ("get_amplitude_get_fptr"))
call c_f_procpointer (c_fptr, prc_lib%get_amplitude_get_fptr)
call process_library_load_configuration (prc_lib, os_data, mdl)
```

```
      prc_lib%status = STAT_LOADED
      if (associated (prc_lib%reload_hook)) &
         call prc_lib%reload_hook (process_library_get_name (prc_lib))
      call var_list_set_string (var_list, var_str ("$library_name"), &
           process_library_get_name (prc_lib), is_known=.true.)  ! $
      if (present (model))  model => mdl
    end subroutine process_library_load
```

Unload a process library. Necessary before recompiling and reloading.

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_unload
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_unload (prc_lib)
      type(process_library_t), intent(inout) :: prc_lib
      call msg_message ("Unloading process library '" // &
           char (process_library_get_name (prc_lib)) // "'")
      if (associated (prc_lib%unload_hook)) &
         call prc_lib%unload_hook (process_library_get_name(prc_lib))
      call dlaccess_final (prc_lib%dlaccess)
      prc_lib%status = STAT_CODE_GENERATED
    end subroutine process_library_unload
```

Register hooks for un- / reloading the process library.

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_set_unload_hook
    public :: process_library_set_reload_hook
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_set_unload_hook (prc_lib, hook)
      type(process_library_t), intent(inout), target :: prc_lib
      procedure(prclib_unload_hook), pointer, intent(in) :: hook
      prc_lib%unload_hook => hook
    end subroutine process_library_set_unload_hook

    subroutine process_library_set_reload_hook (prc_lib, hook)
      type(process_library_t), intent(inout), target :: prc_lib
      procedure(prclib_reload_hook), pointer, intent(in) :: hook
      prc_lib%reload_hook => hook
    end subroutine process_library_set_reload_hook
```

Get a C function pointer to a procedure belonging to the process library interface and check for an error condition.

⟨*Process libraries: procedures*⟩+≡
```
    function process_library_get_c_funptr &
         (prc_lib, prefix, fname) result (c_fptr)
      type(c_funptr) :: c_fptr
      type(process_library_t), intent(inout) :: prc_lib
      type(string_t), intent(in) :: prefix, fname
      type(string_t) :: full_name
      full_name = prefix // "_" // fname
      if (prc_lib%static) then
         c_fptr = libmanager_get_c_funptr (char (prefix), char (fname))
```

```
    else
       c_fptr = dlaccess_get_c_funptr (prc_lib%dlaccess, full_name)
       call process_library_check_dlerror (prc_lib)
    end if
  end function process_library_get_c_funptr
```

Check for an error condition and signal it.

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_check_dlerror (prc_lib)
      type(process_library_t), intent(in) :: prc_lib
      if (dlaccess_has_error (prc_lib%dlaccess)) then
         call msg_fatal (char (dlaccess_get_error (prc_lib%dlaccess)))
      end if
    end subroutine process_library_check_dlerror
```

13.1.14   The library store

We want to handle several libraries in parallel, therefore we introduce a global
library store, similar to the model and process lists. The store is a module
variable.

⟨*Process libraries: types*⟩+≡
```
    type :: process_library_store_t
      private
      type(process_library_t), pointer :: first => null ()
      type(process_library_t), pointer :: last => null ()
    end type process_library_store_t
```

⟨*Process libraries: variables*⟩≡
```
    type(process_library_store_t), save :: process_library_store
```

Append a new library, if it does not yet exist, and return a pointer to it.

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_store_append
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_store_append (name, os_data, prc_lib)
      type(string_t), intent(in) :: name
      type(os_data_t), intent(in) :: os_data
      type(process_library_t), pointer :: prc_lib
      prc_lib => process_library_store_get_ptr (name)
      if (.not. associated (prc_lib)) then
         call msg_message &
              ("Initializing process library '" // char (name) // "'")
         allocate (prc_lib)
         call process_library_init (prc_lib, name, os_data)
         if (associated (process_library_store%last)) then
            process_library_store%last%next => prc_lib
         else
            process_library_store%first => prc_lib
         end if
         process_library_store%last => prc_lib
      end if
```

```
      end subroutine process_library_store_append
```

Finalizer. This closes all open libraries.

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_store_final
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_store_final ()
      type(process_library_t), pointer :: current
      do while (associated (process_library_store%first))
         current => process_library_store%first
         process_library_store%first => current%next
         call process_library_final (current)
         deallocate (current)
      end do
    end subroutine process_library_store_final
```

Load all libraries

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_store_load
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_store_load (os_data, var_list)
      type(os_data_t), intent(in) :: os_data
      type(var_list_t), intent(inout), optional :: var_list
      type(process_library_t), pointer :: current
      current => process_library_store%first
      do while (associated (current))
         call process_library_load (current, os_data, var_list=var_list)
         current => current%next
      end do
    end subroutine process_library_store_load
```

Get a pointer to an existing (named) library

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_store_get_ptr
```

⟨*Process libraries: procedures*⟩+≡
```
    function process_library_store_get_ptr (name) result (prc_lib)
      type(process_library_t), pointer :: prc_lib
      type(string_t), intent(in) :: name
      prc_lib => process_library_store%first
      do while (associated (prc_lib))
         if (prc_lib%basename == name)  exit
         prc_lib => prc_lib%next
      end do
    end function process_library_store_get_ptr
```

Get a pointer to the first/next library

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_store_get_first
```

⟨*Process libraries: procedures*⟩+≡
```
function process_library_store_get_first () result (prc_lib)
  type(process_library_t), pointer :: prc_lib
  prc_lib => process_library_store%first
end function process_library_store_get_first
```

13.1.15 Preloading static libraries

Static libraries are static, so it is sensible to load them all at startup. (By default, they are linked, but not loaded in the sense that a `process_library` object exists for them.) This can be done using this routine.

⟨*Process libraries: public*⟩+≡
```
public :: process_library_store_load_static
```

⟨*Process libraries: procedures*⟩+≡
```
subroutine process_library_store_load_static &
     (os_data, prc_lib, model, var_list)
  type(os_data_t), intent(in) :: os_data
  type(process_library_t), pointer :: prc_lib
  type(model_t), pointer :: model
  type(var_list_t), intent(inout) :: var_list
  integer :: n, i
  type(string_t), dimension(:), allocatable :: libname
  n = libmanager_get_n_libs ()
  allocate (libname (n))
  do i = 1, n
     libname(i) = libmanager_get_libname (i)
  end do
  do i = 1, n
     call process_library_store_append (libname(i), os_data, prc_lib)
     call process_library_set_static (prc_lib, .true.)
     call process_library_load (prc_lib, os_data, model, var_list)
  end do
end subroutine process_library_store_load_static
```

13.1.16 Integration results

Record integration results.

⟨*Process libraries: public*⟩+≡
```
public :: process_library_record_integral
```

⟨*Process libraries: procedures*⟩+≡
```
subroutine process_library_record_integral &
     (prc_lib, prc_id, n_calls, integral, error, accuracy, chi2, efficiency)
  type(process_library_t), intent(inout), target :: prc_lib
  type(string_t), intent(in) :: prc_id
  integer, intent(in) :: n_calls
  real(default), intent(in) :: integral, error, accuracy, chi2, efficiency
  type(process_configuration_t), pointer :: prc_conf
  prc_conf => process_library_get_process_ptr (prc_lib, prc_id)
  if (associated (prc_conf)) then
```

```
        if (prc_conf%status >= STAT_LOADED) then
           call process_configuration_record_integral &
                (prc_conf, n_calls, integral, error, accuracy, chi2, efficiency)
        else
           call msg_bug ("Process '" // char (prc_id) // "': not loaded, " &
                // "can't record integral")
        end if
     else
        call msg_bug ("Process '" // char (prc_id) // "': not associated, " &
             // "can't record integral")
     end if
   end subroutine process_library_record_integral
```

⟨*Process libraries: public*⟩+≡
```
   public :: process_library_get_n_calls
   public :: process_library_get_integral
   public :: process_library_get_error
   public :: process_library_get_accuracy
   public :: process_library_get_chi2
   public :: process_library_get_efficiency
```
⟨*Process libraries: procedures*⟩+≡
```
   function process_library_get_n_calls (prc_lib, prc_id) result (n_calls)
     integer :: n_calls
     type(process_library_t), intent(in), target :: prc_lib
     type(string_t), intent(in) :: prc_id
     type(process_configuration_t), pointer :: prc_conf
     prc_conf => process_library_get_process_ptr (prc_lib, prc_id)
     if (associated (prc_conf)) then
        n_calls = prc_conf%n_calls
     else
        n_calls = 0
     end if
   end function process_library_get_n_calls

   function process_library_get_integral (prc_lib, prc_id) result (integral)
     real(default) :: integral
     type(process_library_t), intent(in), target :: prc_lib
     type(string_t), intent(in) :: prc_id
     type(process_configuration_t), pointer :: prc_conf
     prc_conf => process_library_get_process_ptr (prc_lib, prc_id)
     if (associated (prc_conf)) then
        integral = prc_conf%integral
     else
        integral = 0
     end if
   end function process_library_get_integral

   function process_library_get_error (prc_lib, prc_id) result (error)
     real(default) :: error
     type(process_library_t), intent(in), target :: prc_lib
     type(string_t), intent(in) :: prc_id
     type(process_configuration_t), pointer :: prc_conf
     prc_conf => process_library_get_process_ptr (prc_lib, prc_id)
```

```
      if (associated (prc_conf)) then
         error = prc_conf%error
      else
         error = 0
      end if
    end function process_library_get_error

    function process_library_get_accuracy (prc_lib, prc_id) result (accuracy)
      real(default) :: accuracy
      type(process_library_t), intent(in), target :: prc_lib
      type(string_t), intent(in) :: prc_id
      type(process_configuration_t), pointer :: prc_conf
      prc_conf => process_library_get_process_ptr (prc_lib, prc_id)
      if (associated (prc_conf)) then
         accuracy = prc_conf%accuracy
      else
         accuracy = 0
      end if
    end function process_library_get_accuracy

    function process_library_get_chi2 (prc_lib, prc_id) result (chi2)
      real(default) :: chi2
      type(process_library_t), intent(in), target :: prc_lib
      type(string_t), intent(in) :: prc_id
      type(process_configuration_t), pointer :: prc_conf
      prc_conf => process_library_get_process_ptr (prc_lib, prc_id)
      if (associated (prc_conf)) then
         chi2 = prc_conf%chi2
      else
         chi2 = 0
      end if
    end function process_library_get_chi2

    function process_library_get_efficiency (prc_lib, prc_id) result (efficiency)
      real(default) :: efficiency
      type(process_library_t), intent(in), target :: prc_lib
      type(string_t), intent(in) :: prc_id
      type(process_configuration_t), pointer :: prc_conf
      prc_conf => process_library_get_process_ptr (prc_lib, prc_id)
      if (associated (prc_conf)) then
         efficiency = prc_conf%efficiency
      else
         efficiency = 0
      end if
    end function process_library_get_efficiency
```

13.1.17   Test

⟨*Process libraries: public*⟩+≡
```
    public :: process_libraries_test
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_libraries_test ()
```

```fortran
    type(model_t), pointer :: model
    type(process_library_t), pointer :: prc_lib
    type(string_t), dimension(:), allocatable :: prt_in, prt_out
    type(os_data_t) :: os_data
    type(string_t) :: objlist
    call os_data_init (os_data)
    os_data%fcflags = "-gline -C=all"
    print *, "*** Read model file"
    call syntax_model_file_init ()
    call model_list_read_model &
         (var_str("QCD"), var_str("test.mdl"), os_data, model)
    call syntax_model_file_final ()
    print *, "*** Create library 'proc' with two processes"
    print *, "* Setup process configuration"
    print *, "  [temporary: include zero processes because of references"
    print *, "   to omegalib, which we also need as a .so version]"
    print *, "  [iso_varying_string included in libproc.so for the same reason"
    call process_library_store_append (var_str ("proc"), os_data, prc_lib)
    allocate (prt_in (1), prt_out (2))
    prt_in(1) = "Z"
    prt_out(1) = "e1"
    prt_out(2) = "E1"
    call process_library_append &
         (prc_lib, var_str ("zee"), model, prt_in, prt_out)
    deallocate (prt_in, prt_out)
    allocate (prt_in (2), prt_out (2))
    prt_in(1) = "g"
    prt_in(2) = "g"
    prt_out(1) = "u"
    prt_out(2) = "U"
    call process_library_append &
         (prc_lib, var_str ("uu"), model, prt_in, prt_out)
    print *
    print *, "* Generate code"
    call process_library_generate_code (prc_lib, os_data)
    print *
    print *, "* Write driver file 'proc_interface.f90'"
    call process_library_write_driver (prc_lib)
    print *
    print *, "* Compile and link as 'libproc.so'"
    call process_library_compile (prc_lib, os_data, .false., objlist)
    call process_library_link (prc_lib, os_data, objlist)
    print *
    print *, "* Load shared libraries"
    call process_library_store_load (os_data)
    print *
    print *, "* Execute 'get_n_processes' from the shared library named 'proc'"
    print *
    prc_lib => process_library_store_get_ptr (var_str ("proc"))
    print *, "n_prc = ", prc_lib% get_n_prc ()
    print *
    print *, "* Cleanup"
    call process_library_store_final
end subroutine process_libraries_test
```

13.2 Hard interactions

This module is concerned with the matrix element of an elementary interaction (typically, a hard scattering or heavy-particle decay). The module does not hold phase space information.

⟨`hard_interactions.f90`⟩≡
  ⟨*File header*⟩

```
  module hard_interactions

    use iso_c_binding !NODEP!
    use kinds !NODEP!
```
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use os_interface
    use models
    use flavors
    use helicities
    use colors
    use quantum_numbers
    use interactions
    use evaluators
    use prclib_interfaces
    use process_libraries
```

  ⟨*Standard module head*⟩

  ⟨*Hard interactions: public*⟩

  ⟨*Hard interactions: types*⟩

  ⟨*Hard interactions: interfaces*⟩

```
  contains
```

  ⟨*Hard interactions: procedures*⟩

```
  end module hard_interactions
```

13.2.1 The hard-interaction data type

We define a special data type that accesses the process library. While constant data are stored as data, the process-specific functions for initialization, calculation and finalization are stored as procedure pointers.

⟨*Hard interactions: types*⟩≡
```
    type :: hard_interaction_data_t
      type(string_t) :: id
```

```
      type(model_t), pointer :: model => null ()
      integer :: n_tot = 0
      integer :: n_in = 0
      integer :: n_out = 0
      integer :: n_flv = 0
      integer :: n_hel = 0
      integer :: n_col = 0
      integer :: n_cin = 0
      integer :: n_cf  = 0
      real(default), dimension(:), allocatable :: par
      integer, dimension(:,:), allocatable :: flv_state, hel_state
      integer, dimension(:,:,:), allocatable :: col_state
      logical, dimension(:,:), allocatable :: ghost_flag
      integer, dimension(:,:), allocatable :: col_flow_index
      complex(default), dimension(:), allocatable :: col_factor
      procedure(prc_init), nopass, pointer :: init => null ()
      procedure(prc_final), nopass, pointer :: final => null ()
      procedure(prc_update_alpha_s), nopass, pointer :: update_alpha_s => null ()
      procedure(prc_reset_helicity_selection), nopass, pointer :: &
           reset_helicity_selection => null ()
      procedure(prc_new_event), nopass, pointer :: new_event => null ()
      procedure(prc_is_allowed), nopass, pointer :: is_allowed => null ()
      procedure(prc_get_amplitude), nopass, pointer :: get_amplitude => null ()
    end type hard_interaction_data_t
```

Initialize the hard process, using the process ID and the model parameters.

Assigning flavor/helicity/color tables: we need an intermediate allocatable array to serve as a C pointer target; the C pointer is passed to the process library where it is dereferenced and the array is filled. In principle, this copying step is necessary only if the Fortran and C types differ (which happens for the logical type). However, since this is not critical, we do it anyway.

For incoming particles, the particle color is inverted. This is useful for squaring the color flow, but has to be undone before convoluting with structure functions.

We define additional functions for finalizing and resetting the pointers into the process library when the library is reloaded.

⟨*Hard interactions: procedures*⟩≡
```
    subroutine hard_interaction_data_unload (data)
      type(hard_interaction_data_t), intent(inout) :: data
      call data%final
      nullify (data%init)
      nullify (data%final)
      nullify (data%update_alpha_s)
      nullify (data%reset_helicity_selection)
      nullify (data%new_event)
      nullify (data%is_allowed)
      nullify (data%get_amplitude)
    end subroutine hard_interaction_data_unload

    subroutine hard_interaction_data_reload (data, prc_lib, pid)
      type(hard_interaction_data_t), intent(inout) :: data
      type(process_library_t), intent(in) :: prc_lib
      integer, optional :: pid
```

```
      integer :: the_pid
      type(c_funptr) :: fptr
      if (present (pid)) then
         the_pid = pid
      else
         the_pid = process_library_get_process_pid (prc_lib, data%id)
         if (the_pid <= 0) call msg_bug &
            ("Invalid process ID '" // char (data%id) // "'")
      end if
      call prc_lib% init_get_fptr (the_pid, fptr)
      call c_f_procpointer (fptr, data% init)
      call prc_lib% final_get_fptr (the_pid, fptr)
      call c_f_procpointer (fptr, data% final)
      call prc_lib% update_alpha_s_get_fptr (the_pid, fptr)
      call c_f_procpointer (fptr, data% update_alpha_s)
      call prc_lib% reset_helicity_selection_get_fptr (the_pid, fptr)
      call c_f_procpointer (fptr, data% reset_helicity_selection)
      call prc_lib% new_event_get_fptr (the_pid, fptr)
      call c_f_procpointer (fptr, data% new_event)
      call prc_lib% is_allowed_get_fptr (the_pid, fptr)
      call c_f_procpointer (fptr, data% is_allowed)
      call prc_lib% get_amplitude_get_fptr (the_pid, fptr)
      call c_f_procpointer (fptr, data% get_amplitude)
    end subroutine hard_interaction_data_reload
```

⟨*Hard interactions: procedures*⟩+≡

```
    subroutine hard_interaction_data_init &
         (data, prc_lib, process_index, process_id, model)
      type(hard_interaction_data_t), intent(out) :: data
      type(process_library_t), intent(in) :: prc_lib
      integer, intent(in) :: process_index
      type(string_t), intent(in) :: process_id
      type(model_t), intent(in), target :: model
      integer(c_int) :: pid
      type(string_t) :: model_name
      integer(c_int), dimension(:,:), allocatable, target :: flv_state, hel_state
      integer(c_int), dimension(:,:,:), allocatable, target :: col_state
      logical(c_bool), dimension(:,:), allocatable, target :: ghost_flag
      integer(c_int), dimension(:), allocatable, target :: cf_index1, cf_index2
      complex(c_default_complex), dimension(:), allocatable, target :: col_factor
      integer :: c, i
      if (.not. associated (prc_lib% get_process_id)) then
         call msg_fatal ("Process library '" // char (prc_lib%basename) // "':" &
               // " procedures unavailable (missing compile command?)")
         data%id = ""
         return
      end if
      pid = process_index
      data%id = process_library_get_process_id (prc_lib, pid)
      if (data%id /= process_id) then
         call msg_bug ("Process ID mismatch: requested '" &
               // char (process_id) // "' but found '" // char (data%id) // "'")
      end if
      data%model => model
```

```
model_name = process_library_get_process_model_name (prc_lib, pid)
if (model_get_name (data%model) /= model_name) then
   call msg_warning ("Process '" // char (process_id) // "': " &
         // "temporarily resetting model from '" &
         // char (model_get_name (data%model)) // "' to '" &
         // char (model_name) // "'")
   data%model => model_list_get_model_ptr (model_name)
   if (.not. associated (data%model)) then
      call msg_fatal ("Model '" // char (model_name) &
           // "' is not initialized")
   end if
end if
data%n_in  = prc_lib% get_n_in  (pid)
data%n_out = prc_lib% get_n_out (pid)
data%n_tot = data%n_in + data%n_out
data%n_flv = prc_lib% get_n_flv (pid)
data%n_hel = prc_lib% get_n_hel (pid)
data%n_col = prc_lib% get_n_col (pid)
data%n_cin = prc_lib% get_n_cin (pid)
data%n_cf  = prc_lib% get_n_cf  (pid)
if (data%n_flv == 0) then
   call msg_warning ("Process '" // char (process_id) // "': " &
         // "matrix element vanishes.")
end if
call model_parameters_to_array (data%model, data%par)
allocate (data%flv_state (data%n_tot, data%n_flv))
allocate (data%hel_state (data%n_tot, data%n_hel))
allocate (data%col_state (data%n_cin, data%n_tot, data%n_col))
allocate (data%ghost_flag (data%n_tot, data%n_col))
allocate (data%col_flow_index (2, data%n_cf))
allocate (data%col_factor (data%n_cf))
allocate (flv_state (data%n_tot, data%n_flv))
allocate (hel_state (data%n_tot, data%n_hel))
allocate (col_state (data%n_cin, data%n_tot, data%n_col))
allocate (ghost_flag (data%n_tot, data%n_col))
allocate (cf_index1 (data%n_cf))
allocate (cf_index2 (data%n_cf))
allocate (col_factor (data%n_cf))
call prc_lib% set_flv_state (pid, &
      c_loc (flv_state), &
      int((/data%n_tot, data%n_flv/), kind=c_int))
data%flv_state = flv_state
call prc_lib% set_hel_state (pid, &
      c_loc (hel_state), &
      int((/data%n_tot, data%n_hel/), kind=c_int))
data%hel_state = hel_state
call prc_lib% set_col_state (pid, &
      c_loc (col_state), &
      int((/data%n_cin, data%n_tot, data%n_col/), kind=c_int), &
      c_loc (ghost_flag), &
      int((/data%n_tot, data%n_col/), kind=c_int))
if (data%n_cin /= 2)  &
      call msg_bug ("Process library '" // char (prc_lib%basename) // "':" &
            // " number of color indices must be two")
```

994

```
      forall (c = 1:2, i = 1:data%n_in)
         data%col_state(c,i,:) = - col_state(3-c,i,:)
      end forall
      forall (i = data%n_in+1:data%n_tot)
         data%col_state(:,i,:) = col_state(:,i,:)
      end forall
      data%ghost_flag = ghost_flag
      call prc_lib% set_cf_table (pid, &
           c_loc (cf_index1), c_loc (cf_index2), c_loc (col_factor), &
           int ((/data%n_cf/), kind=c_int))
      data%col_flow_index(1,:) = cf_index1
      data%col_flow_index(2,:) = cf_index2
      data%col_factor = col_factor
      call hard_interaction_data_reload (data, prc_lib, pid=pid)
   end subroutine hard_interaction_data_init
```

I/O:

⟨*Hard interactions: procedures*⟩+≡

```
   subroutine hard_interaction_data_write (data, unit)
     type(hard_interaction_data_t), intent(in) :: data
     integer, intent(in), optional :: unit
     integer :: f, h, c, n, i
     integer :: u
     u = output_unit (unit);  if (u < 0)  return
     write (u, *) "Process '", char (trim (data%id)), "'"
     write (u, *) "n_tot = ", data%n_tot
     write (u, *) "n_in  = ", data%n_in
     write (u, *) "n_out = ", data%n_out
     write (u, *) "n_flv = ", data%n_flv
     write (u, *) "n_hel = ", data%n_hel
     write (u, *) "n_col = ", data%n_col
     write (u, *) "n_cin = ", data%n_cin
     write (u, *) "n_cf  = ", data%n_cf
     write (u, *) "Model parameters:"
     do i = 1, size (data%par)
        write (u, *) i, data%par(i)
     end do
     write (u, *) "Flavor states:"
     do f = 1, data%n_flv
        write (u, *) f, ":", data%flv_state (:,f)
     end do
     write (u, *) "Helicity states:"
     do h = 1, data%n_hel
        write (u, *) h, ":", data%hel_state (:,h)
     end do
     write (u, *) "Color states:"
     do c = 1, data%n_col
        write (u, "(I5,A)", advance="no") c, ":"
        do n = 1, data%n_tot
           write (u, "('/')", advance="no")
           if (data%ghost_flag (n, c))  write (u, "('*')", advance="no")
           do i = 1, data%n_cin
              if (data%col_state(i,n,c) == 0)  cycle
```

995

```
            write (u, "(I3)", advance="no")  data%col_state(i,n,c)
         end do
      end do
      write (u, "('/')")
   end do
   write (u, *) "Color factors:"
   do c = 1, data%n_cf
      write (u, "(I5,A,2(I4,1x))", advance="no") c, ":", &
            data%col_flow_index(:,c)
      write (u, *) data%col_factor(c)
   end do
end subroutine hard_interaction_data_write
```

13.2.2 The hard-interaction type

The type contains an interaction that is used to store the bare matrix element values. The flavor/helicity/color arrays are used to identify each matrix element for the amplitude function. Furthermore, there are three evaluators for the trace (the squared matrix element proper), the squared matrix element with color factors, possibly exclusive in some quantum numbers, and the squared matrix element broken down by color flows. The latter two are needed only for the simulation of complete events, not for integration.

⟨*Hard interactions: public*⟩≡
```
   public :: hard_interaction_t
```

⟨*Hard interactions: types*⟩+≡
```
   type :: hard_interaction_t
     private
     logical :: initialized = .false.
     type(hard_interaction_data_t) :: data
     integer :: n_values = 0
     integer, dimension(:), allocatable :: flv, hel, col
     type(interaction_t) :: int
     type(evaluator_t) :: eval_trace
     type(evaluator_t) :: eval_sqme
     type(evaluator_t) :: eval_flows
   end type hard_interaction_t
```

Initializer. Set up the hard-process data and build the corresponding interaction structure. In parallel, assign the allowed flavor/helicity/color indices to the corresponding index arrays. For each valid combination, a matrix element pointer is prepared which is inserted as a new leaf in the interaction quantum-number tree.

In addition to initialization, we also provide subroutines for partial finalization and re-initialization if the process library is reloaded.

⟨*Hard interactions: public*⟩+≡
```
   public :: hard_interaction_init
   public :: hard_interaction_unload
   public :: hard_interaction_reload
```

⟨*Hard interactions: procedures*⟩+≡
```
    subroutine hard_interaction_unload (hi)
      type(hard_interaction_t), intent(inout), target :: hi
      if (.not. associated (hi%data%final)) return
      call hard_interaction_data_unload (hi%data)
    end subroutine hard_interaction_unload

    subroutine hard_interaction_reload (hi, prc_lib)
      type(hard_interaction_t), intent(inout), target :: hi
      type(process_library_t), intent(in) :: prc_lib
      if (associated (hi%data%init)) return
      call hard_interaction_data_reload (hi%data, prc_lib)
      call hi%data% init (real (hi%data%par, c_default_float))
    end subroutine hard_interaction_reload

    subroutine hard_interaction_init &
         (hi, prc_lib, process_index, process_id, model)
      type(hard_interaction_t), intent(out), target :: hi
      type(process_library_t), intent(in) :: prc_lib
      integer, intent(in) :: process_index
      type(string_t), intent(in) :: process_id
      type(model_t), intent(in), target :: model
      type(flavor_t), dimension(:), allocatable :: flv
      type(color_t), dimension(:), allocatable :: col
      type(helicity_t), dimension(:), allocatable :: hel
      type(quantum_numbers_t), dimension(:), allocatable :: qn
      integer :: f, h, c, i, n
      call hard_interaction_data_init &
           (hi%data, prc_lib, process_index, process_id, model)
      if (hi%data%id == "") return
      call hi%data% init (real (hi%data%par, c_default_float))
      call interaction_init &
           (hi%int, hi%data%n_in, 0, hi%data%n_out, set_relations=.true.)
      call hard_interaction_reset_helicity_selection (hi, 0._default, 0)
      n = 0
      do f = 1, hi%data%n_flv
         do h = 1, hi%data%n_hel
            do c = 1, hi%data%n_col
               if (hi%data%is_allowed (f, h, c))  n = n + 1
            end do
         end do
      end do
      hi%n_values = n
      allocate (hi%flv (n), hi%hel (n), hi%col (n))
      allocate (flv (hi%data%n_tot), col (hi%data%n_tot), hel (hi%data%n_tot))
      allocate (qn (hi%data%n_tot))
      i = 0
      do f = 1, hi%data%n_flv
         do h = 1, hi%data%n_hel
            do c = 1, hi%data%n_col
               if (hi%data%is_allowed (f, h, c)) then
                  i = i + 1
                  hi%flv(i) = f
                  hi%hel(i) = h
```

```
                hi%col(i) = c
                call flavor_init (flv, hi%data%flv_state(:,f), hi%data%model)
                call color_init_from_array (col, hi%data%col_state(:,:,c), &
                                             hi%data%ghost_flag(:,c))
                call helicity_init (hel, hi%data%hel_state(:,h))
                call quantum_numbers_init (qn, flv, col, hel)
                call interaction_add_state (hi%int, qn)
             end if
          end do
       end do
    end do
    call interaction_freeze (hi%int)
    hi%initialized = .true.
  end subroutine hard_interaction_init
```

Finalizer:

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_final
```

⟨*Hard interactions: procedures*⟩+≡
```
    subroutine hard_interaction_final (hi)
      type(hard_interaction_t), intent(inout) :: hi
      hi%initialized = .false.
      if (associated (hi%data% final))  call hi%data% final ()
      call interaction_final (hi%int)
      call evaluator_final (hi%eval_trace)
      call evaluator_final (hi%eval_flows)
      call evaluator_final (hi%eval_sqme)
      hi%n_values = 0
      if (allocated (hi%flv))  deallocate (hi%flv)
      if (allocated (hi%hel))  deallocate (hi%hel)
      if (allocated (hi%col))  deallocate (hi%col)
    end subroutine hard_interaction_final
```

I/O:

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_write
```

⟨*Hard interactions: procedures*⟩+≡
```
    subroutine hard_interaction_write &
        (hi, unit, verbose, show_momentum_sum, show_mass, write_comb)
      type(hard_interaction_t), intent(in) :: hi
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
      logical, intent(in), optional :: write_comb
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,A)")  "Hard interaction:"
      call hard_interaction_data_write (hi%data, u)
      if (present (write_comb)) then
         if (write_comb .and. hi%n_values /= 0) then
            write (u, "(1x,A)")  "Allowed f/h/c index combinations:"
            do i = 1, hi%n_values
               write (u, *)  i, ":", hi%flv(i), hi%hel(i), hi%col(i)
```

```
        end do
      end if
    end if
    write (u, *)
    call interaction_write &
         (hi%int, unit, verbose, show_momentum_sum, show_mass)
    write (u, *) repeat ("- ", 36)
    write (u, "(A)") "Trace including color factors (hard interaction)"
    call evaluator_write &
         (hi%eval_trace, unit, verbose, show_momentum_sum, show_mass)
    write (u, *) repeat ("- ", 36)
    write (u, "(A)") "Exclusive sqme including color factors (hard interaction)"
    call evaluator_write &
         (hi%eval_sqme, unit, verbose, show_momentum_sum, show_mass)
    write (u, *) repeat ("- ", 36)
    write (u, "(A)") "Color flow coefficients (hard interaction)"
    call evaluator_write &
         (hi%eval_flows, unit, verbose, show_momentum_sum, show_mass)
  end subroutine hard_interaction_write
```

Defined assignment. Deep copy (except for procedure pointers, of course).

⟨*Hard interactions: public*⟩+≡
```
    public :: assignment(=)
```

⟨*Hard interactions: interfaces*⟩≡
```
    interface assignment(=)
       module procedure hard_interaction_assign
    end interface
```

⟨*Hard interactions: procedures*⟩+≡
```
    subroutine hard_interaction_assign (hi_out, hi_in)
      type(hard_interaction_t), intent(out) :: hi_out
      type(hard_interaction_t), intent(in) :: hi_in
      hi_out%initialized = hi_in%initialized
      hi_out%data = hi_in%data
      hi_out%n_values = hi_in%n_values
      if (allocated (hi_in%flv)) then
         allocate (hi_out%flv (size (hi_in%flv)))
         hi_out%flv = hi_in%flv
      end if
      if (allocated (hi_in%hel)) then
         allocate (hi_out%hel (size (hi_in%hel)))
         hi_out%hel = hi_in%hel
      end if
      if (allocated (hi_in%col)) then
         allocate (hi_out%col (size (hi_in%col)))
         hi_out%col = hi_in%col
      end if
      hi_out%int = hi_in%int
      hi_out%eval_trace = hi_in%eval_trace
      hi_out%eval_sqme = hi_in%eval_sqme
      hi_out%eval_flows = hi_in%eval_flows
    end subroutine hard_interaction_assign
```

13.2.3 Access contents

Whether we have a valid data set:

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_is_valid
```

⟨*Hard interactions: procedures*⟩+≡
```
    function hard_interaction_is_valid (hi) result (flag)
      logical :: flag
      type(hard_interaction_t), intent(in) :: hi
      flag = hi%initialized
    end function hard_interaction_is_valid
```

The alphanumeric ID.

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_get_id
```

⟨*Hard interactions: procedures*⟩+≡
```
    function hard_interaction_get_id (hi) result (id)
      type(string_t) :: id
      type(hard_interaction_t), intent(in) :: hi
      id = hi%data%id
    end function hard_interaction_get_id
```

The model as used for the hard interaction.

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_get_model_ptr
```

⟨*Hard interactions: procedures*⟩+≡
```
    function hard_interaction_get_model_ptr (hi) result (model)
      type(model_t), pointer :: model
      type(hard_interaction_t), intent(in) :: hi
      model => hi%data%model
    end function hard_interaction_get_model_ptr
```

Particle counts.

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_get_n_in
    public :: hard_interaction_get_n_out
    public :: hard_interaction_get_n_tot
```

⟨*Hard interactions: procedures*⟩+≡
```
    pure function hard_interaction_get_n_in (hi) result (n_in)
      integer :: n_in
      type(hard_interaction_t), intent(in) :: hi
      n_in = hi%data%n_in
    end function hard_interaction_get_n_in

    pure function hard_interaction_get_n_out (hi) result (n_out)
      integer :: n_out
      type(hard_interaction_t), intent(in) :: hi
      n_out = hi%data%n_out
    end function hard_interaction_get_n_out
```

```
    pure function hard_interaction_get_n_tot (hi) result (n_tot)
      integer :: n_tot
      type(hard_interaction_t), intent(in) :: hi
      n_tot = hi%data%n_tot
    end function hard_interaction_get_n_tot
```

Quantum number counts.

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_get_n_flv
    public :: hard_interaction_get_n_col
    public :: hard_interaction_get_n_hel
```

⟨*Hard interactions: procedures*⟩+≡
```
    pure function hard_interaction_get_n_flv (hi) result (n_flv)
      integer :: n_flv
      type(hard_interaction_t), intent(in) :: hi
      n_flv = hi%data%n_flv
    end function hard_interaction_get_n_flv

    pure function hard_interaction_get_n_col (hi) result (n_col)
      integer :: n_col
      type(hard_interaction_t), intent(in) :: hi
      n_col = hi%data%n_col
    end function hard_interaction_get_n_col

    pure function hard_interaction_get_n_hel (hi) result (n_hel)
      integer :: n_hel
      type(hard_interaction_t), intent(in) :: hi
      n_hel = hi%data%n_hel
    end function hard_interaction_get_n_hel
```

Particle tables.

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_get_flv_states
```

⟨*Hard interactions: procedures*⟩+≡
```
    function hard_interaction_get_flv_states (hi) result (flv_state)
      integer, dimension(:,:), allocatable :: flv_state
      type(hard_interaction_t), intent(in) :: hi
      allocate (flv_state (size (hi%data%flv_state, 1), &
                           size (hi%data%flv_state, 2)))
      flv_state = hi%data%flv_state
    end function hard_interaction_get_flv_states
```

Color factor tables.

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_get_n_cf
```

⟨*Hard interactions: procedures*⟩+≡
```
    pure function hard_interaction_get_n_cf (hi) result (n_cf)
      integer :: n_cf
      type(hard_interaction_t), intent(in) :: hi
      n_cf = hi%data%n_cf
    end function hard_interaction_get_n_cf
```

Incoming particles. Consider only the first entry in the array of flavor combinations.

If the process is forbidden and no flavor states are present, create at least an initial state with undefined particles.

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_get_first_pdg_in
```

⟨*Hard interactions: procedures*⟩+≡
```
    function hard_interaction_get_first_pdg_in (hi) result (pdg)
      integer, dimension(:), allocatable :: pdg
      type(hard_interaction_t), intent(in) :: hi
      allocate (pdg (hi%data%n_in))
      if (hi%data%n_flv > 0) then
         pdg = hi%data%flv_state (:hi%data%n_in, 1)
      else
         pdg = 0
      end if
    end function hard_interaction_get_first_pdg_in
```

This procedure is used for checking whether some of the final-state particles can initiate decay cascades. We check only the first row in the flavor array, since unstable particles are massive and should not be subject to flavor summation. Thus they must be common to all rows.

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_get_unstable_products
```

⟨*Hard interactions: procedures*⟩+≡
```
    subroutine hard_interaction_get_unstable_products (hi, flv_unstable)
      type(hard_interaction_t), intent(in) :: hi
      type(flavor_t), dimension(:), intent(out), allocatable :: flv_unstable
      type(model_t), pointer :: model
      integer, dimension(hi%data%n_out) :: pdg_out
      type(flavor_t) :: flv
      integer :: i
      model => hi%data%model
      if (associated (model) .and. size (hi%data%flv_state, 2) /= 0) then
         pdg_out = hi%data%flv_state(hi%data%n_in+1:,1)
         do i = 1, size (pdg_out)
            if (pdg_out(i) /= 0) then
               call flavor_init (flv, pdg_out(i), model)
               if (flavor_is_stable (flv)) then
                  where (pdg_out(i:) == pdg_out(i))  pdg_out(i:) = 0
               else
                  where (pdg_out(i+1:) == pdg_out(i))  pdg_out(i+1:) = 0
               end if
            end if
         end do
         allocate (flv_unstable (count (pdg_out /= 0)))
         call flavor_init &
              (flv_unstable, pack (pdg_out, pdg_out /= 0), model)
      else
         allocate (flv_unstable (0))
      end if
    end subroutine hard_interaction_get_unstable_products
```

13.2.4 Evaluators

This procedure initializes the evaluator that computes the matrix element squared, traced over all outgoing quantum numbers. Whether the trace over incoming quantum numbers is done, depends on the specified mask – except for color which is always summed.

⟨*Hard interactions: public*⟩+≡
```
public :: hard_interaction_init_trace
```

⟨*Hard interactions: procedures*⟩+≡
```
subroutine hard_interaction_init_trace &
    (hi, qn_mask_in, use_hi_color_factors, nc)
  type(hard_interaction_t), intent(inout), target :: hi
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
  logical, intent(in), optional :: use_hi_color_factors
  integer, intent(in), optional :: nc
  logical :: use_hi_cf
  type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
  if (present (use_hi_color_factors)) then
     use_hi_cf = use_hi_color_factors
  else
     use_hi_cf = .false.
  end if
  allocate (qn_mask (hi%data%n_tot))
  qn_mask(:hi%data%n_in) = &
       new_quantum_numbers_mask (.false., .true., .false.) &
       .or. qn_mask_in
  qn_mask(hi%data%n_in+1:) = &
       new_quantum_numbers_mask (.true., .true., .true.)
  if (use_hi_cf) then
     call evaluator_init_square (hi%eval_trace, hi%int, qn_mask, &
          hi%data%col_flow_index, hi%data%col_factor, hi%col, nc=nc)
  else
     call evaluator_init_square (hi%eval_trace, hi%int, qn_mask, nc=nc)
  end if
end subroutine hard_interaction_init_trace
```

This procedure initializes the evaluator that computes the matrix element square separated in parts (e.g., polarization components). Polarization is kept in the initial state (if allowed by `qn_mask_in`) and for those final-state particles which are marked as unstable. The incoming-particle mask can also be used to sum over incoming flavor.

⟨*Hard interactions: public*⟩+≡
```
public :: hard_interaction_init_sqme
```

⟨*Hard interactions: procedures*⟩+≡
```
subroutine hard_interaction_init_sqme &
    (hi, qn_mask_in, use_hi_color_factors, nc)
  type(hard_interaction_t), intent(inout), target :: hi
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
  logical, intent(in), optional :: use_hi_color_factors
```

```
integer, intent(in), optional :: nc
logical :: use_hi_cf
type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
type(flavor_t), dimension(:), allocatable :: flv
integer :: i
if (present (use_hi_color_factors)) then
   use_hi_cf = use_hi_color_factors
else
   use_hi_cf = .false.
end if
allocate (qn_mask (hi%data%n_tot), flv (hi%data%n_flv))
qn_mask(:hi%data%n_in) = &
     new_quantum_numbers_mask (.false., .true., .false.) &
     .or. qn_mask_in
do i = hi%data%n_in + 1, hi%data%n_tot
   call flavor_init (flv, hi%data%flv_state(i,:), hi%data%model)
   qn_mask(i) = new_quantum_numbers_mask (.false., .true., &
           all (flavor_decays_isotropically (flv) &
                .or. flavor_is_stable (flv)), &
           mask_hd = all (flavor_decays_diagonal (flv)))
end do
if (use_hi_cf) then
   call evaluator_init_square (hi%eval_sqme, hi%int, qn_mask, &
        hi%data%col_flow_index, hi%data%col_factor, hi%col, nc=nc)
else
   call evaluator_init_square (hi%eval_sqme, hi%int, qn_mask, nc=nc)
end if
end subroutine hard_interaction_init_sqme
```

This procedure initializes the evaluator that computes the contributions to color flows, neglecting color interference. Polarization is summed over. The incoming-particle mask can be used to sum over incoming flavor.

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_init_flows
```

⟨*Hard interactions: procedures*⟩+≡
```
    subroutine hard_interaction_init_flows (hi, qn_mask_in)
      type(hard_interaction_t), intent(inout), target :: hi
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
      type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
      type(flavor_t), dimension(:), allocatable :: flv
      integer :: i
      allocate (qn_mask (hi%data%n_tot), flv (hi%data%n_flv))
      qn_mask(:hi%data%n_in) = &
           new_quantum_numbers_mask (.false., .false., .false.) &
           .or. qn_mask_in
      do i = hi%data%n_in + 1, hi%data%n_tot
         call flavor_init (flv, hi%data%flv_state(i,:), hi%data%model)
         qn_mask(i) = new_quantum_numbers_mask (.false., .false., &
                 all (flavor_decays_isotropically (flv) &
                      .or. flavor_is_stable (flv)), &
                 mask_hd = all (flavor_decays_diagonal (flv)))
      end do
      call evaluator_init_square (hi%eval_flows, hi%int, qn_mask, &
```

```
          expand_color_flows = .true.)
    end subroutine hard_interaction_init_flows
```

Finalize the previous evaluators.

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_final_sqme
    public :: hard_interaction_final_flows
```

⟨*Hard interactions: procedures*⟩+≡
```
    subroutine hard_interaction_final_sqme (hi)
      type(hard_interaction_t), intent(inout) :: hi
      call evaluator_final (hi%eval_sqme)
    end subroutine hard_interaction_final_sqme

    subroutine hard_interaction_final_flows (hi)
      type(hard_interaction_t), intent(inout) :: hi
      call evaluator_final (hi%eval_flows)
    end subroutine hard_interaction_final_flows
```

13.2.5   Matrix-element evaluation

Update the α_s value used by the matrix element (if any).

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_update_alpha_s
```

⟨*Hard interactions: procedures*⟩+≡
```
    subroutine hard_interaction_update_alpha_s (hi, alpha_s)
      type(hard_interaction_t), intent(inout) :: hi
      real(default), intent(in) :: alpha_s
      real(c_default_float) :: c_alpha_s
      c_alpha_s = alpha_s
      call hi%data% update_alpha_s (c_alpha_s)
    end subroutine hard_interaction_update_alpha_s
```

Reset the helicity selection counters that are used to speed up things by dropping
zero helicity channels after `cutoff` tries.

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_reset_helicity_selection
```

⟨*Hard interactions: procedures*⟩+≡
```
    subroutine hard_interaction_reset_helicity_selection (hi, threshold, cutoff)
      type(hard_interaction_t), intent(inout) :: hi
      real(default), intent(in) :: threshold
      integer, intent(in) :: cutoff
      real(c_default_float) :: c_threshold
      integer(c_int) :: c_cutoff
      c_threshold = threshold
      c_cutoff = cutoff
      call hi%data% reset_helicity_selection (c_threshold, c_cutoff)
    end subroutine hard_interaction_reset_helicity_selection
```

This interfaces the matrix element proper. First, we request a new matrix element value to be computed from the given momenta. Then, we extract all values that are known to be allowed and assign them to the matrix element array. This array consists of pointers to the interaction values, so in fact the latter are calculated.

Although it may be irrelevant, this is an obvious place for parallel execution, so write a forall assignment. Making the assignment elemental is not possible because `get_amplitude` is a procedure pointer. [This is deactivated; to be checked again.]

After the matrix element data are read, we evaluate the squared matrix element (`eval_trace`). square and the color-flow coefficients.

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_evaluate
```

⟨*Hard interactions: procedures*⟩+≡
```
    subroutine hard_interaction_evaluate (hi)
      type(hard_interaction_t), intent(inout), target :: hi
      integer :: i
      complex(default) :: val
      call hi%data% new_event &
          (array_from_vector4 (interaction_get_momenta (hi%int)))
!     forall (i = 1:hi%n_values)
!        hi%me(i) = hi%data% get_amplitude (hi%flv(i), hi%hel(i), hi%col(i))
!     end forall
      do i = 1, hi%n_values
         val = hi%data% get_amplitude (hi%flv(i), hi%hel(i), hi%col(i))
         call interaction_set_matrix_element (hi%int, i, val)
      end do
      call evaluator_evaluate (hi%eval_trace)
    end subroutine hard_interaction_evaluate
```

The extra evaluators (squared matrix element without trace, color flows) need only be evaluated for simulation events that pass the unweighting step. This follows the previous routine.

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_evaluate_sqme
    public :: hard_interaction_evaluate_flows
```

⟨*Hard interactions: procedures*⟩+≡
```
    subroutine hard_interaction_evaluate_sqme (hi)
      type(hard_interaction_t), intent(inout), target :: hi
      call evaluator_receive_momenta (hi%eval_sqme)
      call evaluator_evaluate (hi%eval_sqme)
    end subroutine hard_interaction_evaluate_sqme

    subroutine hard_interaction_evaluate_flows (hi)
      type(hard_interaction_t), intent(inout), target :: hi
      call evaluator_receive_momenta (hi%eval_flows)
      call evaluator_evaluate (hi%eval_flows)
    end subroutine hard_interaction_evaluate_flows
```

This provides direct access to the matrix element, squared and traced over all
quantum numbers. It is not used for ordinary evaluation.

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_compute_sqme_sum
```

⟨*Hard interactions: procedures*⟩+≡
```
    function hard_interaction_compute_sqme_sum (hi, p) result (sqme)
      real(default) :: sqme
      type(hard_interaction_t), intent(inout), target :: hi
      type(vector4_t), dimension(:), intent(in) :: p
      call interaction_set_momenta (hi%int, p)
      call hard_interaction_evaluate (hi)
      sqme = evaluator_sum (hi%eval_trace)
    end function hard_interaction_compute_sqme_sum
```

13.2.6 Access results

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_get_int_ptr
```

⟨*Hard interactions: procedures*⟩+≡
```
    function hard_interaction_get_int_ptr (hi) result (int)
      type(interaction_t), pointer :: int
      type(hard_interaction_t), intent(in), target :: hi
      int => hi%int
    end function hard_interaction_get_int_ptr
```

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_get_eval_trace_ptr
    public :: hard_interaction_get_eval_sqme_ptr
    public :: hard_interaction_get_eval_flows_ptr
```

⟨*Hard interactions: procedures*⟩+≡
```
    function hard_interaction_get_eval_trace_ptr (hi) result (eval)
      type(evaluator_t), pointer :: eval
      type(hard_interaction_t), intent(in), target :: hi
      eval => hi%eval_trace
    end function hard_interaction_get_eval_trace_ptr

    function hard_interaction_get_eval_sqme_ptr (hi) result (eval)
      type(evaluator_t), pointer :: eval
      type(hard_interaction_t), intent(in), target :: hi
      eval => hi%eval_sqme
    end function hard_interaction_get_eval_sqme_ptr

    function hard_interaction_get_eval_flows_ptr (hi) result (eval)
      type(evaluator_t), pointer :: eval
      type(hard_interaction_t), intent(in), target :: hi
      eval => hi%eval_flows
    end function hard_interaction_get_eval_flows_ptr
```

13.2.7 Test

⟨*Hard interactions: public*⟩+≡
```
    public :: hard_interaction_test
```

⟨*Hard interactions: procedures*⟩+≡
```
    subroutine hard_interaction_test (model)
      type(model_t), pointer :: model
      type(process_library_t) :: prc_lib
      type(os_data_t) :: os_data
      type(hard_interaction_t), target :: hi
      type(vector4_t), dimension(4) :: p
      type(quantum_numbers_mask_t), dimension(2) :: qn_mask_in
      type(quantum_numbers_mask_t), dimension(4) :: qn_mask
      real(default) :: sqme, mh
      call os_data_init (os_data)
      call msg_message ("*** Load library 'qedtest'")
      call msg_message ("    [must exist and contain process 'eemm' (whizard.sin.qedtest)]")
      call process_library_init (prc_lib, var_str("qedtest"), os_data)
      call process_library_load (prc_lib, os_data)
      call msg_message ()
      call msg_message ("*** Create hard interaction")
      call hard_interaction_init (hi, prc_lib, 1, var_str ("eemm"), model)
      qn_mask_in = new_quantum_numbers_mask (.true., .true., .true.)
      call hard_interaction_init_trace (hi, qn_mask_in)
      print *, "Interaction: n_values = ", interaction_get_n_matrix_elements (hi%int)
      qn_mask_in = new_quantum_numbers_mask (.false., .false., .false., .true.)
      call hard_interaction_init_sqme (hi, qn_mask_in)
      call hard_interaction_init_flows (hi, qn_mask_in)
      p(1) = vector4_moving (250._default, 250._default, 3)
      p(2) = vector4_moving (250._default,-250._default, 3)
      p(3) = rotation (1._default, 1) * p(1)
      p(4) = p(1) + p(2) - p(3)
      call msg_message ()
      call msg_message ("*** Evaluate new event")
      sqme = hard_interaction_compute_sqme_sum (hi, p)
      call hard_interaction_evaluate_sqme (hi)
      call hard_interaction_evaluate_flows (hi)
      call hard_interaction_write (hi)
      print *
      print *, "sqme sum =", sqme
      print *
      print *, "*** Cleanup"
      call hard_interaction_final (hi)
      call process_library_final (prc_lib)
    end subroutine hard_interaction_test
```

13.3 Processes

This module combines hard interactions, phase space, and (for scatterings) structure functions and interfaces them to the VAMP integration module.

⟨`processes.f90`⟩≡

⟨File header⟩

```
module processes
```

⟨Use kinds⟩
⟨Use strings⟩
```
  use system_dependencies !NODEP!
  use constants !NODEP!
```
⟨Use file utils⟩
```
  use diagnostics !NODEP!
  use sm_physics !NODEP!
  use vamp_equivalences !NODEP!
  use vamp !NODEP!
  use md5
  use clock
  use os_interface
  use lexers
  use parser
  use lorentz !NODEP!
  use prt_lists
  use variables
  use expressions
  use models
  use flavors
  use quantum_numbers
  use polarizations
  use interactions
  use evaluators
  use particles
  use beams
  use sf_isr
  use sf_epa
  use sf_ewa
  use sf_lhapdf
  use strfun
  use mappings
  use phs_forests
  use cascades
  use process_libraries
  use prclib_interfaces
  use hard_interactions
```

⟨Standard module head⟩

⟨Processes: public⟩

⟨Processes: parameters⟩

⟨Processes: types⟩

⟨Processes: variables⟩

⟨Processes: interfaces⟩

```
    contains

⟨Processes: procedures⟩

    end module processes
```

13.3.1   Integration results

This object collects the results of an integration pass and makes them available
to the outside.

The results object has to distinguish the process type:

⟨*Processes: parameters*⟩≡
```
    integer, parameter :: PRC_UNKNOWN = 0
    integer, parameter :: PRC_DECAY = 1
    integer, parameter :: PRC_SCATTERING = 2
```

We store the process type, the index of the integration pass and the absolute
iteration index, the number of iterations contained in this result (for averages),
and the integral (cross section or partial width), error estimate, efficiency and
time estimate.

For intermediate results, we set a flag if this result is an improvement w.r.t.
previous ones.

⟨*Processes: types*⟩≡
```
    type :: integration_entry_t
       private
       integer :: process_type = PRC_UNKNOWN
       integer :: pass = 0
       integer :: it = 0
       integer :: n_it = 0
       integer :: n_calls = 0
       logical :: improved = .false.
       real(default) :: integral = 0
       real(default) :: error = 0
       real(default) :: efficiency = 0
       real(default) :: chi2 = 0
       real(default), dimension(:), allocatable :: grove_weight
       logical :: time_set = .false.
       type(time_t) :: time_start
       type(time_t) :: time_end
    end type integration_entry_t
```

Initialize with all relevant data

⟨*Processes: procedures*⟩≡
```
    subroutine integration_entry_init (entry, &
         process_type, pass, it, n_it, n_calls, improved, &
         integral, error, efficiency, chi2, grove_weight, &
         time_start, time_end)
      type(integration_entry_t), intent(out) :: entry
      integer, intent(in) :: process_type, pass, it, n_it, n_calls
      logical, intent(in) :: improved
      real(default), intent(in) :: integral, error, efficiency
      real(default), intent(in), optional :: chi2
```

1010

```
      real(default), dimension(:), intent(in), optional :: grove_weight
      type(time_t), intent(in), optional :: time_start, time_end
      integer :: n_groves
      entry%process_type = process_type
      entry%pass = pass
      entry%it = it
      entry%n_it = n_it
      entry%n_calls = n_calls
      entry%improved = improved
      entry%integral = integral
      entry%error = error
      entry%efficiency = efficiency
      if (present (chi2)) &
           entry%chi2 = chi2
      if (present (grove_weight)) then
         n_groves = size (grove_weight)
         allocate (entry%grove_weight (n_groves))
         entry%grove_weight = grove_weight
      end if
      if (present (time_start) .and. present (time_end)) then
         entry%time_start = time_start
         entry%time_end = time_end
         entry%time_set = .true.
      end if
    end subroutine integration_entry_init
```

Access values, some of them computed on demand:

⟨*Processes: procedures*⟩+≡

```
    function integration_entry_get_n_calls (entry) result (n)
      integer :: n
      type(integration_entry_t), intent(in) :: entry
      n = entry%n_calls
    end function integration_entry_get_n_calls

    function integration_entry_get_integral (entry) result (int)
      real(default) :: int
      type(integration_entry_t), intent(in) :: entry
      int = entry%integral
    end function integration_entry_get_integral

    function integration_entry_get_error (entry) result (err)
      real(default) :: err
      type(integration_entry_t), intent(in) :: entry
      err = entry%error
    end function integration_entry_get_error

    function integration_entry_get_relative_error (entry) result (err)
      real(default) :: err
      type(integration_entry_t), intent(in) :: entry
      if (entry%integral /= 0) then
         err = entry%error / entry%integral
      else
         err = 0
      end if
```

```
    end function integration_entry_get_relative_error

  function integration_entry_get_accuracy (entry) result (acc)
    real(default) :: acc
    type(integration_entry_t), intent(in) :: entry
    acc = accuracy (entry%integral, entry%error, entry%n_calls)
  end function integration_entry_get_accuracy

  function accuracy (integral, error, n_calls) result (acc)
    real(default) :: acc
    real(default), intent(in) :: integral, error
    integer, intent(in) :: n_calls
    if (integral /= 0) then
       acc = error / integral * sqrt (real (n_calls, default))
    else
       acc = 0
    end if
  end function accuracy

  function integration_entry_get_efficiency (entry) result (eff)
    real(default) :: eff
    type(integration_entry_t), intent(in) :: entry
    eff = entry%efficiency
  end function integration_entry_get_efficiency

  function integration_entry_get_chi2 (entry) result (chi2)
    real(default) :: chi2
    type(integration_entry_t), intent(in) :: entry
    chi2 = entry%chi2
  end function integration_entry_get_chi2

  function integration_entry_get_time_per_event (entry) result (tpe)
    real(default) :: tpe
    type(integration_entry_t), intent(in) :: entry
    if (entry%time_set) then
       if (entry%n_calls /= 0 .and. entry%efficiency /= 0) then
          tpe = time_to_seconds (entry%time_end - entry%time_start) &
               / entry%n_calls / entry%efficiency
       else
          tpe = 0
       end if
    else
       tpe = 0
    end if
  end function integration_entry_get_time_per_event

  function integration_entry_has_improved (entry) result (flag)
    logical :: flag
    type(integration_entry_t), intent(in) :: entry
    flag = entry%improved
  end function integration_entry_has_improved

  function integration_entry_get_n_groves (entry) result (n_groves)
    integer :: n_groves
```

```
      type(integration_entry_t), intent(in) :: entry
      if (allocated (entry%grove_weight)) then
         n_groves = size (entry%grove_weight)
      else
         n_groves = 0
      end if
    end function integration_entry_get_n_groves
```

Output. This writes the header line for the result account below:

⟨*Processes: procedures*⟩+≡
```
    subroutine write_header (process_type, unit, logfile)
      integer, intent(in) :: process_type
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: logfile
      character(5) :: phys_unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      select case (process_type)
      case (PRC_DECAY);       phys_unit = "[GeV]"
      case (PRC_SCATTERING); phys_unit = "[fb] "
      case default
         phys_unit = ""
      end select
      write (msg_buffer, "(A)") &
           "It      Calls  Integral" // phys_unit // &
           " Error" // phys_unit // &
           "  Err[%]    Acc  Eff[%]   Chi2 N[It] |"
      call msg_message (unit=u, logfile=logfile)
    end subroutine write_header
```

This writes a separator for result display:

⟨*Processes: procedures*⟩+≡
```
    subroutine write_hline (unit)
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)")  "|" // (repeat ("-", 77)) // "|"
    end subroutine write_hline

    subroutine write_dline (unit)
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)")  "|" // (repeat ("=", 77)) // "|"
    end subroutine write_dline
```

This writes the standard result account into one screen line. The verbose version
uses multiple lines and prints the unabridged values.

⟨*Processes: procedures*⟩+≡
```
    subroutine integration_entry_write (entry, unit, verbose)
      type(integration_entry_t), intent(in) :: entry
      integer, intent(in), optional :: unit
```

```fortran
      logical, intent(in), optional :: verbose
      integer :: u
      character(1) :: star
      logical :: verb
      u = output_unit (unit);  if (u < 0)  return
      verb = .false.;  if (present (verbose))  verb = verbose
      if (.not. verb)  then
         if (entry%improved) then
            star = "*"
         else
            star = " "
         end if
1        format (1x, I3, 1x, I10, 1x, 1PE14.7, 1x, 1PE9.2, 1x, 2PF7.2, &
               1x, 0PF7.2, A1, 1x, 2PF6.2, 1x, 0PF7.2, 1x, I3)
         if (entry%n_it /= 1) then
            write (u, 1) &
                 entry%it, &
                 entry%n_calls, &
                 entry%integral, &
                 entry%error, &
                 integration_entry_get_relative_error (entry), &
                 integration_entry_get_accuracy (entry), &
                 star, &
                 entry%efficiency, &
                 entry%chi2, &
                 entry%n_it
         else
            write (u, 1) &
                 entry%it, &
                 entry%n_calls, &
                 entry%integral, &
                 entry%error, &
                 integration_entry_get_relative_error (entry), &
                 integration_entry_get_accuracy (entry), &
                 star, &
                 entry%efficiency
         end if
      else
         write (u, *)  "process_type = ", entry%process_type
         write (u, *)  "        pass = ", entry%pass
         write (u, *)  "          it = ", entry%it
         write (u, *)  "        n_it = ", entry%n_it
         write (u, *)  "     n_calls = ", entry%n_calls
         write (u, *)  "    improved = ", entry%improved
         write (u, *)  "    integral = ", entry%integral
         write (u, *)  "       error = ", entry%error
         write (u, *)  "  efficiency = ", entry%efficiency
         write (u, *)  "        chi2 = ", entry%chi2
         if (allocated (entry%grove_weight)) then
            write (u, *)  "    n_groves = ", size (entry%grove_weight)
            write (u, *)  "grove_weight = ", entry%grove_weight
         else
            write (u, *)  "    n_groves = 0"
         end if
```

```
      end if
    end subroutine integration_entry_write
```

Read the entry, assuming it has been written in verbose format.

⟨Processes: procedures⟩+≡
```
    subroutine integration_entry_read (entry, unit)
      type(integration_entry_t), intent(out) :: entry
      integer, intent(in) :: unit
      character(30) :: dummy
      character :: equals
      integer :: n_groves
      read (unit, *)  dummy, equals, entry%process_type
      read (unit, *)  dummy, equals, entry%pass
      read (unit, *)  dummy, equals, entry%it
      read (unit, *)  dummy, equals, entry%n_it
      read (unit, *)  dummy, equals, entry%n_calls
      read (unit, *)  dummy, equals, entry%improved
      read (unit, *)  dummy, equals, entry%integral
      read (unit, *)  dummy, equals, entry%error
      read (unit, *)  dummy, equals, entry%efficiency
      read (unit, *)  dummy, equals, entry%chi2
      read (unit, *)  dummy, equals, n_groves
      if (n_groves /= 0) then
         allocate (entry%grove_weight (n_groves))
         read (unit, *)  dummy, equals, entry%grove_weight
      end if
    end subroutine integration_entry_read
```

Write an account of the channel weights, accumulated by groves.

⟨Processes: procedures⟩+≡
```
    subroutine integration_entry_write_grove_weights (entry, unit)
      type(integration_entry_t), intent(in) :: entry
      integer, intent(in), optional :: unit
      integer :: n_groves
      character(20) :: fmt
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      if (allocated (entry%grove_weight)) then
         n_groves = size (entry%grove_weight)
         write (fmt, "(A,I0,A)")  "(", n_groves, "(1x,I3))"
         write (u, fmt)  nint (entry%grove_weight * 100)
      end if
    !   contains
    !     function get_ifmt (n) result (fmt)
    !       character(2) :: fmt
    !       integer, intent(in) :: n
    !       character(20) :: tmp_str
    !       integer :: ilen
    !       write (tmp_str, "(I0)")  n
    !       ilen = len_trim (tmp_str)
    !       write (fmt, "(A1,I1)")  "I", ilen
    !     end function get_ifmt
    end subroutine integration_entry_write_grove_weights
```

Compute the average for all entries in the specified integration pass. The integrals are weighted w.r.t. their individual errors, and we compute average, error of the average, and χ^2 value. All errors are assumed Gaussian, of course. The efficiency returned is the one of the last entry in the integration pass. For the time estimate, we simply take the minimum of all which should correspond to the best efficiency.

If any integral or error vanishes, averaging fails.

⟨*Processes: procedures*⟩+≡
```
function compute_average (entry, pass) result (result)
  type(integration_entry_t) :: result
  type(integration_entry_t), dimension(:), intent(in) :: entry
  integer, intent(in) :: pass
  integer :: i
  logical, dimension(size(entry)) :: mask
  real(default), dimension(size(entry)) :: ivar
  real(default) :: sum_ivar, variance
  result%process_type = entry(1)%process_type
  mask = entry%pass == pass
  result%it = maxval (entry%it, mask)
  result%n_it = count (mask)
  result%n_calls = sum (entry%n_calls, mask)
  where (entry%error /= 0)
     ivar = 1 / entry%error ** 2
  elsewhere
     ivar = 0
  end where
  sum_ivar = sum (ivar, mask)
  if (sum_ivar /= 0) then
     variance = 1 / sum_ivar
  else
     variance = 0
  end if
  result%integral = sum (entry%integral * ivar, mask) * variance
  result%error = sqrt (variance)
  if (result%n_it > 1) then
     result%chi2 = sum ((entry%integral - result%integral)**2 * ivar, mask) &
                 / (result%n_it - 1)
  end if
  do i = size (entry), 1, -1
     if (mask(i)) then
        result%efficiency = entry(i)%efficiency
        exit
     end if
  end do
end function compute_average
```

13.3.2 Combined integration results

We collect a list of results which grows during the execution of the program. This is implemented as an array which grows if necessary; so we can easily

compute averages.

⟨Processes: public⟩≡
```
    public :: integration_results_t
```

⟨Processes: types⟩+≡
```
    type :: integration_results_t
       private
       integer :: n_pass = 0
       integer :: n_it = 0
       type(integration_entry_t), dimension(:), allocatable :: entry
       type(integration_entry_t), dimension(:), allocatable :: average
    end type integration_results_t
```

The array is extended in chunks of 10 entries.

⟨Processes: parameters⟩+≡
```
    integer, parameter :: RESULTS_CHUNK_SIZE = 10
```

The standard does not require to explicitly initialize the integers; however, some
gfortran version has a bug here and misses the default initialization in the type
definition.

⟨Processes: procedures⟩+≡
```
    subroutine integration_results_init (results)
      type(integration_results_t), intent(out) :: results
      results%n_pass = 0
      results%n_it = 0
      allocate (results%entry (RESULTS_CHUNK_SIZE))
      allocate (results%average (RESULTS_CHUNK_SIZE))
    end subroutine integration_results_init
```

Output (ASCII format).

⟨Processes: procedures⟩+≡
```
    subroutine integration_results_write (results, unit, verbose)
      type(integration_results_t), intent(in) :: results
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      logical :: verb
      integer :: u, n
      real(default) :: time_per_event
      u = output_unit (unit);  if (u < 0)  return
      verb = .false.;  if (present (verbose))  verb = verbose
      if (.not. verb) then
         call write_dline (unit)
         if (results%n_it /= 0) then
            call write_header (results%entry(1)%pass, unit)
            call write_dline (unit)
            do n = 1, results%n_it
               if (n > 1) then
                  if (results%entry(n)%pass /= results%entry(n-1)%pass) then
                     call write_hline (unit)
                     call integration_entry_write &
                          (results%average(results%entry(n-1)%pass), unit)
                     call write_hline (unit)
```

1017

```
                end if
             end if
             call integration_entry_write (results%entry(n), unit)
          end do
          call write_dline(unit)
          call integration_entry_write (results%average(results%n_pass), unit)
          call write_dline(unit)
       else
          call msg_message ("[WHIZARD integration results: empty]", unit)
       end if
       call write_dline (unit)
    else
       write (u, *)  "begin(integration_results)"
       write (u, *)  "  n_pass = ", results%n_pass
       write (u, *)  "    n_it = ", results%n_it
       if (results%n_it > 0) then
          write (u, *)  "begin(integration_pass)"
          do n = 1, results%n_it
             if (n > 1) then
                if (results%entry(n)%pass /= results%entry(n-1)%pass) then
                   write (u, *)  "end(integration_pass)"
                   write (u, *)  "begin(integration_pass)"
                end if
             end if
             write (u, *)  "begin(iteration)"
             call integration_entry_write (results%entry(n), unit, verb)
             write (u, *)  "end(iteration)"
          end do
          write (u, *)  "end(integration_pass)"
       end if
       write (u, *)  "end(integration_results)"
    end if
  end subroutine integration_results_write
```

Incremental output: Write specific / last line. separator if appropriate.

⟨*Processes: procedures*⟩+≡
```
    subroutine integration_results_write_entry (results, it, unit)
      type(integration_results_t), intent(in) :: results
      integer, intent(in) :: it
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      if (it /= 0)  call integration_entry_write (results%entry(it), unit)
    end subroutine integration_results_write_entry

    subroutine integration_results_write_current (results, unit)
      type(integration_results_t), intent(in) :: results
      integer, intent(in), optional :: unit
      integer :: u, n
      u = output_unit (unit);  if (u < 0)  return
      n = results%n_it
      if (n /= 0)  call integration_entry_write (results%entry(n), unit)
    end subroutine integration_results_write_current
```

Write one line for the average

⟨*Processes: procedures*⟩+≡

```
subroutine integration_results_write_average (results, pass, unit)
  type(integration_results_t), intent(in) :: results
  integer, intent(in) :: pass
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  if (pass /= 0)  call integration_entry_write (results%average(pass), unit)
end subroutine integration_results_write_average

subroutine integration_results_write_current_average (results, unit)
  type(integration_results_t), intent(in) :: results
  integer, intent(in), optional :: unit
  integer :: u, n
  u = output_unit (unit);  if (u < 0)  return
  n = results%n_pass
  if (n /= 0)  call integration_entry_write (results%average(n), unit)
end subroutine integration_results_write_current_average
```

Write a concise table of grove weights, i.e., the channel history.

⟨*Processes: procedures*⟩+≡

```
subroutine integration_results_write_grove_weights (results, unit)
  type(integration_results_t), intent(in) :: results
  integer, intent(in), optional :: unit
  integer :: u, i, n
  u = output_unit (unit);  if (u < 0)  return
  if (results%n_it /= 0) then
     call msg_message ("Phase-space grove weight history: " &
          // "(numbers in %)", unit)
     write (u, "(A9)", advance="no")  "| grove |"
        do i = 1, integration_entry_get_n_groves (results%entry(1))
        write (u, "(1x,I3)", advance="no")  i
     end do
     write (u, *)
     call write_dline (unit)
     do n = 1, results%n_it
        if (n > 1) then
           if (results%entry(n)%pass /= results%entry(n-1)%pass) then
              call write_hline (unit)
           end if
        end if
        write (u, "(1x,I6,1x,A1)", advance="no")  n, "|"
        call integration_entry_write_grove_weights (results%entry(n), unit)
     end do
  else
     call msg_message ("Channel weight history: [undefined]", unit)
  end if
  call write_dline(unit)
end subroutine integration_results_write_grove_weights
```

Read the list from file. The file must be written using the `verbose` option of the writing routine.

⟨*Processes: procedures*⟩+≡
```
    subroutine integration_results_read (results, unit)
      type(integration_results_t), intent(out) :: results
      integer, intent(in) :: unit
      character(80) :: buffer
      character :: equals
      integer :: pass, it
      read (unit, *)  buffer
      if (trim (adjustl (buffer)) /= "begin(integration_results)") then
         call read_err ();  return
      end if
      read (unit, *)  buffer, equals, results%n_pass
      read (unit, *)  buffer, equals, results%n_it
      allocate (results%entry (results%n_it + RESULTS_CHUNK_SIZE))
      allocate (results%average (results%n_it + RESULTS_CHUNK_SIZE))
      it = 0
      do pass = 1, results%n_pass
         read (unit, *)  buffer
         if (trim (adjustl (buffer)) /= "begin(integration_pass)") then
            call read_err ();  return
         end if
         READ_ENTRIES: do
            read (unit, *)  buffer
            if (trim (adjustl (buffer)) /= "begin(iteration)") then
               exit READ_ENTRIES
            end if
            it = it + 1
            call integration_entry_read (results%entry(it), unit)
            read (unit, *)  buffer
            if (trim (adjustl (buffer)) /= "end(iteration)") then
               call read_err (); return
            end if
         end do READ_ENTRIES
         if (trim (adjustl (buffer)) /= "end(integration_pass)") then
            call read_err (); return
         end if
         results%average(pass) = compute_average (results%entry, pass)
      end do
      read (unit, *)  buffer
      if (trim (adjustl (buffer)) /= "end(integration_results)") then
         call read_err (); return
      end if
    contains
      subroutine read_err ()
        call msg_fatal ("Reading integration results from file: syntax error")
      end subroutine read_err
    end subroutine integration_results_read
```

Check integration results for consistency. We compare against an array of pass
indices and call numbers. If there is a difference, up to the number of iterations
done so far, we return failure.

⟨*Processes: procedures*⟩+≡
```
    function integration_results_iterations_are_consistent &
```

```
      (results, pass, n_calls) result (flag)
    logical :: flag
    type(integration_results_t), intent(in) :: results
    integer, dimension(:), intent(in) :: pass, n_calls
    flag = all (results%entry(:results%n_it)%pass == pass(:results%n_it)) &
        .and. all (results%entry(:results%n_it)%n_calls &
                == n_calls(:results%n_it))
  end function integration_results_iterations_are_consistent
```

Expand the list of entries if the limit has been reached:
⟨*Processes: procedures*⟩+≡
```
  subroutine integration_results_expand (results)
    type(integration_results_t), intent(inout) :: results
    type(integration_entry_t), dimension(:), allocatable :: entry_tmp
    if (results%n_it == size (results%entry)) then
       allocate (entry_tmp (results%n_it))
       entry_tmp = results%entry
       deallocate (results%entry)
       allocate (results%entry (results%n_it + RESULTS_CHUNK_SIZE))
       results%entry(:results%n_it) = entry_tmp
       deallocate (entry_tmp)
    end if
    if (results%n_pass == size (results%average)) then
       allocate (entry_tmp (results%n_pass))
       entry_tmp = results%average
       deallocate (results%average)
       allocate (results%average (results%n_it + RESULTS_CHUNK_SIZE))
       results%average(:results%n_pass) = entry_tmp
       deallocate (entry_tmp)
    end if
  end subroutine integration_results_expand
```

Append a new entry to the list and, if appropriate, compute the average.
⟨*Processes: procedures*⟩+≡
```
  subroutine integration_results_append_entry (results, entry)
    type(integration_results_t), intent(inout) :: results
    type(integration_entry_t), intent(in) :: entry
    if (results%n_it == 0) then
       call integration_results_init (results)
       results%n_it = 1
       results%n_pass = 1
    else
       call integration_results_expand (results)
       if (entry%pass /= results%entry(results%n_it)%pass) &
            results%n_pass = results%n_pass + 1
       results%n_it = results%n_it + 1
    end if
    results%entry(results%n_it) = entry
    results%average(results%n_pass) = &
        compute_average (results%entry, entry%pass)
  end subroutine integration_results_append_entry
```

Enter results into the results list.

⟨*Processes: public*⟩+≡
```
public :: integration_results_append
```

⟨*Processes: procedures*⟩+≡
```
subroutine integration_results_append (results, &
    process_type, pass, n_it, n_calls, &
    integral, error, efficiency, grove_weight, time_start, time_end)
  type(integration_results_t), intent(inout) :: results
  integer, intent(in) :: process_type, pass, n_it, n_calls
  real(default), intent(in) :: integral, error, efficiency
  real(default), dimension(:), intent(in), optional :: grove_weight
  type(time_t), intent(in), optional :: time_start, time_end
  logical :: improved
  type(integration_entry_t) :: entry
  if (results%n_it /= 0) then
     improved = accuracy (integral, error, n_calls) &
           < integration_entry_get_accuracy (results%entry(results%n_it))
  else
     improved = .true.
  end if
  call integration_entry_init (entry, &
        process_type, pass, results%n_it+1, n_it, n_calls, improved, &
        integral, error, efficiency, grove_weight=grove_weight, &
        time_start=time_start, time_end=time_end)
  call integration_results_append_entry (results, entry)
end subroutine integration_results_append
```

13.3.3   Access results

Get the last average of the integral.

⟨*Processes: procedures*⟩+≡
```
function integration_results_get_n_calls (results) result (n_calls)
  integer :: n_calls
  type(integration_results_t), intent(in) :: results
  if (results%n_pass > 0) then
     n_calls = &
           integration_entry_get_n_calls (results%average(results%n_pass))
  else
     n_calls = 0
  end if
end function integration_results_get_n_calls

function integration_results_get_integral (results) result (integral)
  real(default) :: integral
  type(integration_results_t), intent(in) :: results
  if (results%n_pass > 0) then
     integral = &
           integration_entry_get_integral (results%average(results%n_pass))
  else
     integral = 0
  end if
end function integration_results_get_integral
```

```
function integration_results_get_error (results) result (error)
  real(default) :: error
  type(integration_results_t), intent(in) :: results
  if (results%n_pass > 0) then
     error = &
          integration_entry_get_error (results%average(results%n_pass))
  else
     error = 0
  end if
end function integration_results_get_error

function integration_results_get_accuracy (results) result (accuracy)
  real(default) :: accuracy
  type(integration_results_t), intent(in) :: results
  if (results%n_pass > 0) then
     accuracy = &
          integration_entry_get_accuracy (results%average(results%n_pass))
  else
     accuracy = 0
  end if
end function integration_results_get_accuracy

function integration_results_get_chi2 (results) result (chi2)
  real(default) :: chi2
  type(integration_results_t), intent(in) :: results
  if (results%n_pass > 0) then
     chi2 = &
          integration_entry_get_chi2 (results%average(results%n_pass))
  else
     chi2 = 0
  end if
end function integration_results_get_chi2

function integration_results_get_efficiency (results) result (efficiency)
  real(default) :: efficiency
  type(integration_results_t), intent(in) :: results
  if (results%n_pass > 0) then
     efficiency = &
          integration_entry_get_efficiency (results%average(results%n_pass))
  else
     efficiency = 0
  end if
end function integration_results_get_efficiency
```

Return the time per event for the last integration pass.

⟨*Processes: procedures*⟩+≡

```
function integration_results_get_time_per_event (results) result (s)
  real(default) :: s
  type(integration_results_t), intent(in) :: results
  if (results%n_pass /= 0) then
     s = integration_entry_get_time_per_event (results%entry(results%n_pass))
  else
     s = 0
```

```
      end if
   end function integration_results_get_time_per_event
```

Return the last pass index and the index of the last iteration *within* the last pass.

⟨*Processes: procedures*⟩+≡

```
   function integration_results_get_current_pass (results) result (pass)
     integer :: pass
     type(integration_results_t), intent(in) :: results
     pass = results%n_pass
   end function integration_results_get_current_pass

   function integration_results_get_current_it (results) result (it)
     integer :: it
     type(integration_results_t), intent(in) :: results
     if (allocated (results%entry)) then
        it = count (results%entry%pass == results%n_pass)
     else
        it = 0
     end if
   end function integration_results_get_current_it
```

Compute the MD5 sum by printing everything and checksumming the resulting file.

⟨*Processes: procedures*⟩+≡

```
   function integration_results_get_md5sum (results) result (md5sum_results)
     character(32) :: md5sum_results
     type(integration_results_t), intent(in) :: results
     integer :: u
     u = free_unit ()
     open (unit = u, status = "scratch", action = "readwrite")
     call integration_results_write (results, u, verbose=.true.)
     rewind (u)
     md5sum_results = md5sum (u)
     close (u)
   end function integration_results_get_md5sum
```

13.3.4   The process type

The process object holds virtually everything that is connected to a particular process; it is the workspace for integration and event generation.

Each process has a type (decay or scattering). For the purpose of generating cascades, we may need multiple copies of a particular (decay) process, which are implemented as a linked list. These store copies of, e.g., the hard-interaction object, the phase space and the VAMP grids. For read-only information they point back to the original.

⟨*Processes: public*⟩+≡

```
   public :: process_t
```

⟨*Processes: types*⟩+≡

```
    type :: process_t
      private
      integer :: type = PRC_UNKNOWN
      type(process_t), pointer :: copy => null ()
      logical :: is_original = .true.
      type(process_t), pointer :: original => null ()
      type(process_t), pointer :: working_copy => null ()
      logical :: in_use = .true.
      logical :: initialized = .false.
      logical :: has_matrix_element = .false.
      logical :: use_hi_color_factors = .false.
      logical :: use_beams = .true.
      logical :: has_extra_evaluators = .true.
      logical :: beams_are_set = .false.
      type(flavor_t), dimension(:), allocatable :: flv_in
      type(beam_data_t) :: beam_data
      type(string_t) :: id
      character(32) :: md5sum = ""
      type(process_library_t), pointer :: prc_lib => null ()
      integer :: lib_index = 0
      integer :: store_index = 0
      type(model_t), pointer :: model
      integer :: n_strfun = 0
      integer :: n_par_strfun = 0
      integer :: n_par_hi = 0
      integer :: n_par = 0
      logical :: azimuthal_dependence = .false.
      logical :: vamp_grids_defined = .false.
      logical :: sqrts_known = .false.
      logical :: sqrts_hat_known = .false.
      real(default) :: sqrts = 0
      real(default) :: sqrts_hat = 0
      real(default), dimension(:), allocatable :: x_strfun
      real(default), dimension(:), allocatable :: x_hi
      integer :: n_channels = 0
      integer :: n_bins = 0
      integer :: channel = 0
      logical :: lab_is_cm_frame = .true.
      type(lorentz_transformation_t) :: lt_cm_to_lab = identity
      real(default), dimension(:,:), allocatable :: x
      real(default), dimension(:), allocatable :: phs_factor
      real(default), dimension(:), allocatable :: mass_in
      real(default) :: flux_factor = 0
      real(default) :: averaging_factor = 0
      real(default) :: sf_mapping_factor = 0
      real(default) :: phs_volume = 0
      real(default) :: vamp_phs_factor = 0
      real(default) :: sqme = 0
      real(default) :: reweighting_factor = 0
      real(default) :: sample_function_value = 0
      real(default) :: scale = 0
      logical :: alpha_s_is_fixed = .true.
      integer :: alpha_s_order = 0
```

```
      integer :: alpha_s_nf = 0
      logical :: alpha_s_from_mz = .true.
      logical :: mz_is_known = .false.
      real(default) :: mz = 0
      logical :: alpha_s_mz_is_known = .false.
      real(default) :: alpha_s_mz = 0
      real(default) :: lambda_qcd = 0
      real(default) :: alpha_s_at_scale = 0
      logical :: alpha_s_from_lhapdf = .false.
      type(strfun_chain_t) :: sfchain
      type(hard_interaction_t) :: hi
      type(evaluator_t) :: eval_trace
      type(evaluator_t) :: eval_beam_flows
      type(evaluator_t) :: eval_sqme
      type(evaluator_t) :: eval_flows
      logical :: fatal_beam_decay = .true.
      type(phs_forest_t) :: forest
      character(32) :: md5sum_phs = ""
      type(vamp_equivalences_t) :: vamp_eq
      type(prt_list_t) :: prt_list
      type(var_list_t) :: var_list
      type(eval_tree_t) :: cut_expr
      type(eval_tree_t) :: reweighting_expr
      type(eval_tree_t) :: scale_expr
      logical, dimension(:), allocatable :: active_channel
      type(vamp_grids) :: grids
      type(vamp_history), dimension(:), allocatable :: v_history
      type(vamp_history), dimension(:,:), allocatable :: v_histories
      type(integration_results_t) :: results
    end type process_t
```

Initialization. We set up the hard-interaction parameters and make them available in the variable list (which extends the variable list of the current model). Finally, we initialize the particle list that is used for evaluating expressions.

The flag `use_beams` may be set false. In that case, beam (and structure function) data are meaningless or are skipped. For a scattering process, a head-to-head collision is assumed. For a decay process, the particle is assumed to decay in its rest frame. The initial state is assumed unpolarized.

If a variable list is provided as an argument, it replaces the model variable list. This implies that it should be linked to the model variable list.

⟨*Processes: procedures*⟩+≡
```
    subroutine process_init &
        (process, prc_lib, process_lib_index, process_store_index, &
         process_id, model, lhapdf_status, var_list, use_beams)
      type(process_t), intent(out), target :: process
      type(process_library_t), intent(in), target :: prc_lib
      integer, intent(in) :: process_lib_index
      integer, intent(in) :: process_store_index
      type(string_t), intent(in) :: process_id
      type(model_t), intent(in), target :: model
      type(lhapdf_status_t), intent(inout) :: lhapdf_status
      type(var_list_t), intent(in), optional, target :: var_list
      logical, intent(in), optional :: use_beams
```

```
integer :: n_in, n_out, n_tot
integer :: lhapdf_set, lhapdf_member
type(string_t) :: lhapdf_prefix, lhapdf_file
type(var_list_t), pointer :: var_list_snapshot
process%prc_lib => prc_lib
process%lib_index = process_lib_index
process%store_index = process_store_index
process%id = process_id
process%md5sum = process_library_get_process_md5sum &
     (process%prc_lib, process%lib_index)
call hard_interaction_init &
     (process%hi, prc_lib, process_lib_index, process_id, model)
process%has_matrix_element = hard_interaction_get_n_flv (process%hi) /= 0
process%use_hi_color_factors = &
     var_list_get_lval (var_list, var_str ("?read_color_factors"))
process%model => hard_interaction_get_model_ptr (process%hi)
if (.not. hard_interaction_is_valid (process%hi)) then
   return
else
   process%id = hard_interaction_get_id (process%hi)
   if (.not. process%has_matrix_element) then
     process%initialized = .true.
     return
   end if
end if
if (present (use_beams)) then
   process%use_beams = use_beams
   process%has_extra_evaluators = use_beams
end if
n_in  = hard_interaction_get_n_in  (process%hi)
n_out = hard_interaction_get_n_out (process%hi)
n_tot = hard_interaction_get_n_tot (process%hi)
select case (n_in)
case (1);  process%type = PRC_DECAY
case (2);  process%type = PRC_SCATTERING
end select
allocate (process%flv_in (n_in), process%mass_in (n_in))
call flavor_init (process%flv_in, &
     hard_interaction_get_first_pdg_in (process%hi), process%model)
process%mass_in = flavor_get_mass (process%flv_in)
if (process%use_beams) then
   process%averaging_factor = 1
else
   process%averaging_factor = &
        1._default / product (flavor_get_multiplicity (process%flv_in))
end if
allocate (var_list_snapshot)
call var_list_link (process%var_list, var_list_snapshot)
if (present (var_list)) then
   call var_list_init_snapshot (var_list_snapshot, var_list)
else
   call var_list_init_snapshot (var_list_snapshot, &
        model_get_var_list_ptr (process%model))
end if
```

```
process%alpha_s_is_fixed = &
     var_list_get_lval (process%var_list, var_str ("?alpha_s_is_fixed"))
process%alpha_s_order = &
     var_list_get_ival (process%var_list, var_str ("alpha_s_order"))
process%alpha_s_nf = &
     var_list_get_ival (process%var_list, var_str ("alpha_s_nf"))
process%alpha_s_from_mz = &
     var_list_get_lval (process%var_list, var_str ("?alpha_s_from_mz"))
process%alpha_s_from_lhapdf = &
     var_list_get_lval (process%var_list, var_str ("?alpha_s_from_lhapdf"))
process%fatal_beam_decay = &
     var_list_get_lval (process%var_list, var_str ("?fatal_beam_decay"))
if (process%alpha_s_from_lhapdf) then
   if (LHAPDF_AVAILABLE) then
      lhapdf_set = 1
      lhapdf_prefix = LHAPDF_PDFSETS_PATH // "/"
      lhapdf_file = var_list_get_sval (var_list, &
          var_str ("$lhapdf_file"))  ! $
      lhapdf_member = var_list_get_ival (var_list, &
          var_str ("lhapdf_member"))
      call lhapdf_init (lhapdf_status, &
          lhapdf_set, lhapdf_prefix, lhapdf_file, lhapdf_member)
   else
      call msg_error &
          ("LHAPDF not linked: reset alpha_s_from_lhapdf to false")
      process%alpha_s_from_lhapdf = .false.
   end if
end if
process%mz_is_known = &
     var_list_is_known (process%var_list, var_str ("mZ"))
if (process%mz_is_known)  process%mz = &
     var_list_get_rval (process%var_list, var_str ("mZ"))
process%alpha_s_mz_is_known = &
     var_list_is_known (process%var_list, var_str ("alphas"))
if (process%alpha_s_mz_is_known)  process%alpha_s_mz = &
     var_list_get_rval (process%var_list, var_str ("alphas"))
process%lambda_qcd = &
     var_list_get_rval (process%var_list, var_str ("lambda_qcd"))
call var_list_append_int (process%var_list, &
     var_str ("n_in"),  n_in, intrinsic=.true.)
call var_list_append_int (process%var_list, &
     var_str ("n_out"), n_out, intrinsic=.true.)
call var_list_append_int (process%var_list, &
     var_str ("n_tot"), n_tot, intrinsic=.true.)
call var_list_append_real_ptr (process%var_list, &
     var_str ("sqrts"), process%sqrts, process%sqrts_known, &
     intrinsic=.true.)
call var_list_append_real_ptr (process%var_list, &
     var_str ("sqrts_hat"), process%sqrts_hat, process%sqrts_hat_known, &
     intrinsic=.true.)
call interaction_init_prt_list &
     (hard_interaction_get_int_ptr (process%hi), process%prt_list)
call integration_results_init (process%results)
process%initialized = .true.
```

```
      end subroutine process_init
```

Finalization. In process copies, some components are just pointers to the original, so they should not be finalized separately.

⟨*Processes: procedures*⟩+≡
```
    recursive subroutine process_final (process)
      type(process_t), intent(inout), target :: process
      call process_delete_copies (process)
      process%initialized = .false.
      process%type = PRC_UNKNOWN
      process%sqrts_known = .false.
      process%sqrts_hat_known = .false.
      call strfun_chain_final (process%sfchain)
      call hard_interaction_final (process%hi)
      call evaluator_final (process%eval_trace)
      call evaluator_final (process%eval_beam_flows)
      call evaluator_final (process%eval_sqme)
      call evaluator_final (process%eval_flows)
      call phs_forest_final (process%forest)
      call vamp_equivalences_final (process%vamp_eq)
      if (process%is_original) then
         call var_list_final (process%var_list)
         call eval_tree_final (process%cut_expr)
         call eval_tree_final (process%reweighting_expr)
         call eval_tree_final (process%scale_expr)
      end if
      if (process%vamp_grids_defined) then
         call vamp_delete_grids (process%grids)
      end if
      call process_final_vamp_history (process)
    end subroutine process_final
```

Output. This prints lots of stuff. The `verbose` option is for state matrices, the `show_momentum_sum` option prints the sums of incoming and outgoing momenta for all interactions, and the `show_mass` option computes and prints the signed invariant mass for all four-momenta.

⟨*Processes: public*⟩+≡
```
    public :: process_write
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_write &
        (process, unit, verbose, show_momentum_sum, show_mass)
      type(process_t), intent(in) :: process
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)")  repeat ("=", 72)
      write (u, *)  "Process data:", process%lib_index, &
          "(", char (process%id), ")"
      select case (process%type)
      case (PRC_UNKNOWN);    write (u, *) "  [unknown]"
      case (PRC_DECAY);      write (u, *) "  [decay]"
```

```
case (PRC_SCATTERING);  write (u, *) "  [scattering]"
end select
write (u, *)  "  use separate beam setup = ", process%use_beams
call beam_data_write (process%beam_data, u)
if (process%use_beams) then
   write (u, *)  "  number of structure functions  = ", process%n_strfun
   write (u, *)  "  number of strfun parameters    = ", process%n_par_strfun
end if
write (u, *)  "  number of process parameters   = ", process%n_par_hi
write (u, *)  "  number of parameters total     = ", process%n_par
write (u, *)  "  number of integration channels = ", process%n_channels
write (u, *)  "  number of bins per channel     = ", process%n_bins
if (process%sqrts_known) then
   write (u, *)  "  c.m. energy (sqrts)     = ", process%sqrts
else
   write (u, *)  "  c.m. energy (sqrts)     = [unknown]"
end if
if (process%sqrts_hat_known) then
   write (u, *)  "  c.m. energy (sqrts_hat) = ", process%sqrts_hat
else
   write (u, *)  "  c.m. energy (sqrts_hat) = [unknown]"
end if
write (u, "(1x,A)", advance="no")  "  Colliding partons       = "
if (allocated (process%flv_in)) then
   do i = 1, size (process%flv_in)
      if (i == 2)  write (u, "(1x)", advance="no")
      call flavor_write (process%flv_in(i), u)
   end do
   write (u, *)
else
   write (u, *)  "[undefined]"
end if
if (allocated (process%mass_in)) then
   write (u, *)  "  Incoming parton masses  = ", process%mass_in
else
   write (u, *)  "  Incoming parton masses  = [unknown]"
end if
write (u, *)  "  In-state flux factor    = ", process%flux_factor
write (u, *)  "  Strfun mapping factor   = ", process%sf_mapping_factor
if (.not. process%use_beams) then
   write (u, *)  "  Spin averaging factor   = ", process%averaging_factor
end if
write (u, *)  "  VAMP phs factor         = ", process%vamp_phs_factor
write (u, *)  "  Phase-space volume      = ", process%phs_volume
write (u, *)  "  Squared matrix element  = ", process%sqme
write (u, *)  "  Reweighting factor      = ", process%reweighting_factor
write (u, *)  "  Sample-function value   = ", &
     process%sample_function_value
write (u, *)  repeat ("-", 72)
if (process%use_beams) then
   write (u, *)  "Structure function parameters ="
   if (allocated (process%x_strfun)) then
      write (u, *)  process%x_strfun
   else
```

```
            write (u, *)  "[empty]"
         end if
      end if
      write (u, *)  "  Process energy scale    = ", process%scale
      write (u, *)  repeat ("-", 72)
      write (u, *)  "QCD coupling parameters ="
      write (u, *)  "  alpha-s is fixed = ", process%alpha_s_is_fixed
      if (.not. process%alpha_s_is_fixed) then
         if (process%alpha_s_from_lhapdf) then
            write (u, *)  "  alpha-s from         LHAPDF"
         else
            write (u, *)  "  LLA order        = ", process%alpha_s_order
            write (u, *)  "  active flavors   = ", process%alpha_s_nf
            write (u, *)  "  use alpha-s (mZ) = ", process%alpha_s_from_mz
            if (process%alpha_s_from_mz) then
               write (u, *)  "  mZ is known      = ", process%mz_is_known
               if (process%mz_is_known) then
                  write (u, *)  "  mZ               = ", process%mz
               end if
               write (u, *)  "  as(mZ) is known  = ", process%alpha_s_mz_is_known
               if (process%alpha_s_mz_is_known) then
                  write (u, *)  "  alpha-s (mZ)     = ", process%alpha_s_mz
               end if
            else
               write (u, *)  "  Lambda_QCD       = ", process%lambda_qcd
            end if
         end if
         write (u, *)  "  alpha-s (scale)  = ", process%alpha_s_at_scale
      end if
      write (u, *)  repeat ("-", 72)
      write (u, *)  "Phase-space integration parameters (input) ="
      if (allocated (process%x_hi)) then
         write (u, *)  process%x_hi
      else
         write (u, *)  "[empty]"
      end if
      write (u, *)  "Integration channel =", process%channel
      write (u, *)  "Phase-space integration parameters (complete) ="
      if (allocated (process%x)) then
         do i = 1, size (process%x, 2)
            write (u, *)  process%x(:,i)
         end do
      else
         write (u, *)  "[empty]"
      end if
      if (.not. process%lab_is_cm_frame) then
         write (u, *)  "Tranformation c.m. -> lab ="
         call lorentz_transformation_write (process%lt_cm_to_lab, u)
      end if
      write (u, *)  "Channels: phase-space factors ="
      if (allocated (process%phs_factor)) then
         write (u, *)  process%phs_factor
      else
         write (u, *) "[not allocated]"
```

```
      end if
      write (u, "(A)")  repeat ("-", 72)
      if (process%use_beams) then
         call strfun_chain_write &
              (process%sfchain, unit, verbose, show_momentum_sum, show_mass)
         write (u, "(A)")  repeat ("-", 72)
         write (u, "(A)") "Incoming beams with all color contractions"
         call evaluator_write &
              (process%eval_beam_flows, unit, verbose, show_momentum_sum, show_mass)
         write (u, "(A)")  repeat ("-", 72)
      end if
      call hard_interaction_write &
           (process%hi, unit, verbose, show_momentum_sum, show_mass)
      write (u, "(A)")  repeat ("-", 72)
      if (process%has_extra_evaluators) then
         write (u, "(A)") "Trace including color factors (beams + strfun + hard interaction)"
         call evaluator_write &
              (process%eval_trace, unit, verbose, show_momentum_sum, show_mass)
         write (u, "(A)")  repeat ("-", 72)
         write (u, "(A)") "Exclusive sqme including color factors (beams + strfun + hard interaction
         call evaluator_write &
              (process%eval_sqme, unit, verbose, show_momentum_sum, show_mass)
         write (u, "(A)")  repeat ("-", 72)
         write (u, "(A)") "Color flow coefficients (beams + strfun + hard interaction)"
         call evaluator_write &
              (process%eval_flows, unit, verbose, show_momentum_sum, show_mass)
         write (u, "(A)")  repeat ("-", 72)
      end if
      call phs_forest_write (process%forest, unit)
      write (u, "(A)")  repeat ("-", 72)
      call vamp_equivalences_write (process%vamp_eq, unit)
      write (u, "(A)")  repeat ("-", 72)
      call prt_list_write (process%prt_list, unit)
      write (u, "(A)")  repeat ("-", 72)
      call var_list_write (process%var_list, unit)
      write (u, "(A)")  repeat ("-", 72)
      write (u, "(A)")  "Cut expression:"
      call eval_tree_write (process%cut_expr, unit)
      write (u, "(A)")  repeat ("-", 72)
      write (u, "(A)")  "Weight expression:"
      call eval_tree_write (process%reweighting_expr, unit)
      write (u, "(A)")  repeat ("-", 72)
      write (u, "(A)")  "Scale expression:"
      call eval_tree_write (process%scale_expr, unit)
      write (u, "(A)")  repeat ("-", 72)
      if (process%vamp_grids_defined) then
         call vamp_write_grids (process%grids, u)
      else
         write (u, "(A)")  "VAMP grids: [empty]"
      end if
      write (u, "(A)")  repeat ("-", 72)
      if (allocated (process%v_history)) then
         call msg_message (" Global history [vamp]:", unit=u)
         call vamp_write_history (u, process%v_history)
```

1032

```
      else
         call msg_message (" Global history [vamp]: [undefined]", unit=u)
      end if
      write (u, "(A)")  repeat ("-", 72)
      if (allocated (process%v_histories)) then
         call msg_message (" Channel histories [vamp]:", unit=u)
         call vamp_write_history (u, process%v_histories)
      else
         call msg_message (" Channel histories [vamp]: [undefined]", unit=u)
      end if
      write (u, *)
      call integration_results_write (process%results, unit)
      call integration_results_write_grove_weights (process%results, unit)
    end subroutine process_write
```

This version is intended for the logfile. We don't write internal data.

⟨*Processes: procedures*⟩+≡
```
    subroutine process_write_log (process, unit)
      type(process_t), intent(in) :: process
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(A)")  repeat ("#", 79)
      write (u, *)  "Process ID = '" // char (process%id) // "'"
      write (u, "(A)")  repeat ("#", 79)
      write (u, *)  "Integral  = ", process_get_integral (process)
      write (u, *)  "Error     = ", process_get_error (process)
      write (u, *)  "Accuracy  = ", process_get_accuracy (process)
      write (u, *)  "Chi2      = ", process_get_chi2 (process)
      write (u, *)  "Efficiency = ", process_get_efficiency (process)
      write (u, *)  "Time/evt  = ", process_get_time_per_event (process)
      call integration_results_write (process%results, unit)
      call integration_results_write_grove_weights (process%results, unit)
      write (u, "(A)")  repeat ("#", 79)
      call beam_data_write (process%beam_data, u)
      write (u, "(A)")  repeat ("#", 79)
      write (u, "(A)")  "Cut expression:"
      call eval_tree_write (process%cut_expr, unit)
      write (u, "(A)")  repeat ("-", 79)
      write (u, "(A)")  "Weight expression:"
      call eval_tree_write (process%reweighting_expr, unit)
      write (u, "(A)")  repeat ("-", 79)
      write (u, "(A)")  "Scale expression:"
      call eval_tree_write (process%scale_expr, unit)
      write (u, "(A)")  repeat ("#", 79)
      if (allocated (process%v_history)) then
         call msg_message ("Global history [vamp]:", unit=u)
         call vamp_write_history (u, process%v_history)
      else
         call msg_message ("Global history [vamp]: [undefined]", unit=u)
      end if
      write (u, "(A)")  repeat ("-", 72)
      if (allocated (process%v_histories)) then
```

1033

```
      call msg_message ("Channel histories [vamp]:", unit=u)
      call vamp_write_history (u, process%v_histories)
   else
      call msg_message ("Channel histories [vamp]: [undefined]", unit=u)
   end if
   write (u, "(A)")  repeat ("-", 72)
   call msg_message ("Equivalences between channels", unit=u)
   call vamp_equivalences_write (process%vamp_eq, unit=u)
   write (u, "(A)")  repeat ("#", 79)
   write (u, "(A)")  "Variable list:"
   call var_list_write (process%var_list, unit)
   write (u, "(A)")  repeat ("#", 79)
 end subroutine process_write_log
```

Write a complete logfile (with hardcoded name based on the process ID).

⟨*Processes: public*⟩+≡
```
   public :: process_write_logfile
```

⟨*Processes: procedures*⟩+≡
```
   subroutine process_write_logfile (process)
     type(process_t), intent(in) :: process
     type(string_t) :: filename
     integer :: unit
     unit = free_unit ()
     filename = process%id // ".log"
     open (unit = unit, file = char (filename), action = "write", &
           status = "replace")
     call process_write_log (process, unit)
     close (unit)
   end subroutine process_write_logfile
```

13.3.5 Accessing contents

Check if the process has been successfully initialized:

⟨*Processes: public*⟩+≡
```
   public :: process_is_valid
```

⟨*Processes: procedures*⟩+≡
```
   function process_is_valid (process) result (flag)
     logical :: flag
     type(process_t), intent(in) :: process
     flag = process%initialized
   end function process_is_valid
```

Check if the process has a nonvanishing matrix element:

⟨*Processes: public*⟩+≡
```
   public :: process_has_matrix_element
```

⟨*Processes: procedures*⟩+≡
```
   function process_has_matrix_element (process) result (flag)
     logical :: flag
     type(process_t), intent(in) :: process
     flag = process%has_matrix_element
```

```
      end function process_has_matrix_element
```

Return the process ID.

⟨Processes: public⟩+≡
```
    public :: process_get_id
```

⟨Processes: procedures⟩+≡
```
    function process_get_id (process) result (process_id)
      type(string_t) :: process_id
      type(process_t), intent(in) :: process
      process_id = process%id
    end function process_get_id
```

Return the index in the process library:

⟨Processes: public⟩+≡
```
    public :: process_get_lib_index
```

⟨Processes: procedures⟩+≡
```
    function process_get_lib_index (process) result (index)
      integer :: index
      type(process_t), intent(in) :: process
      index = process%lib_index
    end function process_get_lib_index
```

Return the index in the process store:

⟨Processes: public⟩+≡
```
    public :: process_get_store_index
```

⟨Processes: procedures⟩+≡
```
    function process_get_store_index (process) result (index)
      integer :: index
      type(process_t), intent(in) :: process
      index = process%store_index
    end function process_get_store_index
```

Return the MD5 sum of the process configuration.

⟨Processes: public⟩+≡
```
    public :: process_get_md5sum
    public :: process_get_md5sum_parameters
    public :: process_get_md5sum_results
```

⟨Processes: procedures⟩+≡
```
    function process_get_md5sum (process) result (md5sum)
      character(32) :: md5sum
      type(process_t), intent(in) :: process
      md5sum = process%md5sum
    end function process_get_md5sum

    function process_get_md5sum_parameters (process) result (md5sum)
      character(32) :: md5sum
      type(process_t), intent(in) :: process
      md5sum = model_get_parameters_md5sum (process%model)
    end function process_get_md5sum_parameters
```

```
function process_get_md5sum_results (process) result (md5sum)
  character(32) :: md5sum
  type(process_t), intent(in) :: process
  md5sum = integration_results_get_md5sum (process%results)
end function process_get_md5sum_results
```

Return the model pointer.

⟨*Processes: public*⟩+≡
```
public :: process_get_model_ptr
```

⟨*Processes: procedures*⟩+≡
```
function process_get_model_ptr (process) result (model)
  type(model_t), pointer :: model
  type(process_t), intent(in) :: process
  model => process%model
end function process_get_model_ptr
```

Return the number of incoming partons for the hard interaction

⟨*Processes: public*⟩+≡
```
public :: process_get_n_in
public :: process_get_n_out
public :: process_get_n_tot
public :: process_get_n_flv
```

⟨*Processes: procedures*⟩+≡
```
pure function process_get_n_in (process) result (n)
  integer :: n
  type(process_t), intent(in) :: process
  n = hard_interaction_get_n_in (process%hi)
end function process_get_n_in

pure function process_get_n_out (process) result (n)
  integer :: n
  type(process_t), intent(in) :: process
  n = hard_interaction_get_n_out (process%hi)
end function process_get_n_out

pure function process_get_n_tot (process) result (n)
  integer :: n
  type(process_t), intent(in) :: process
  n = hard_interaction_get_n_tot (process%hi)
end function process_get_n_tot

pure function process_get_n_flv (process) result (n)
  integer :: n
  type(process_t), intent(in) :: process
  n = hard_interaction_get_n_flv (process%hi)
end function process_get_n_flv
```

Return the indices of incoming beams / partons. These indices apply to the particle lists in the evaluator interactions.

Without allocate-on-assignment, let us use subroutines.

⟨*Processes: public*⟩+≡
```
    public :: process_get_beam_index
    public :: process_get_incoming_parton_index
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_get_beam_index (process, index)
      type(process_t), intent(in) :: process
      integer, dimension(:), allocatable, intent(out) :: index
      if (process%use_beams) then
         allocate (index (process_get_n_in (process)))
         select case (size (index))
         case (1);  index = (/ 1 /)
         case (2);  index = (/ 1, 2 /)
         end select
      end if
    end subroutine process_get_beam_index

    subroutine process_get_incoming_parton_index (process, index)
      type(process_t), intent(in) :: process
      integer, dimension(:), allocatable, intent(out) :: index
      allocate (index (process_get_n_in (process)))
      select case (size (index))
      case (1);  index = (/ 1 /)
      case (2);  index = (/ 1, 2 /)
      end select
      if (process%use_beams) &
           index = index + strfun_chain_get_n_vir (process%sfchain)
    end subroutine process_get_incoming_parton_index
```

Return the beam/incoming particle flavors and energies.

⟨*Processes: public*⟩+≡
```
    public :: process_get_beam_flv
    public :: process_get_beam_energy
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_beam_flv (process) result (flv_in)
      type(flavor_t), dimension(:), allocatable :: flv_in
      type(process_t), intent(in) :: process
      allocate (flv_in (process_get_n_in (process)))
      if (process%beam_data%initialized)  flv_in = process%beam_data%flv
    end function process_get_beam_flv

    function process_get_beam_energy (process) result (energy)
      real(default), dimension(:), allocatable :: energy
      type(process_t), intent(in) :: process
      allocate (energy (process_get_n_in (process)))
      energy = beam_data_get_energy (process%beam_data)
    end function process_get_beam_energy
```

Return the number of integration parameters.

⟨*Processes: public*⟩+≡
```
    public :: process_get_n_parameters
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_n_parameters (process) result (n)
      integer :: n
      type(process_t), intent(in) :: process
      n = process%n_par
    end function process_get_n_parameters
```

Return the number of integration channels.
⟨*Processes: public*⟩+≡
```
    public :: process_get_n_channels
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_n_channels (process) result (n)
      integer :: n
      type(process_t), intent(in) :: process
      n = process%n_channels
    end function process_get_n_channels
```

Return the number of bins per integration channel.
⟨*Processes: public*⟩+≡
```
    public :: process_get_n_bins
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_n_bins (process) result (n)
      integer :: n
      type(process_t), intent(in) :: process
      n = process%n_bins
    end function process_get_n_bins
```

Return the process energy scale and the α_s value.
⟨*Processes: public*⟩+≡
```
    public :: process_get_scale
    public :: process_get_alpha_s
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_scale (process) result (scale)
      real(default) :: scale
      type(process_t), intent(in) :: process
      scale = process%scale
    end function process_get_scale

    function process_get_alpha_s (process) result (alpha_s)
      real(default) :: alpha_s
      type(process_t), intent(in) :: process
      alpha_s = process%alpha_s_at_scale
    end function process_get_alpha_s
```

Return the squared matrix element. This includes structure function factors and the hard interaction squared matrix element, traced over all quantum numbers, but no phase space factors.
⟨*Processes: public*⟩+≡
```
    public :: process_get_sqme
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_sqme (process) result (sqme)
      real(default) :: sqme
      type(process_t), intent(in) :: process
      sqme = process%sqme
    end function process_get_sqme
```

Return the final integration results.

⟨*Processes: public*⟩+≡
```
    public :: process_get_n_calls
    public :: process_get_integral
    public :: process_get_error
    public :: process_get_accuracy
    public :: process_get_chi2
    public :: process_get_time_per_event
    public :: process_get_efficiency
    public :: process_get_sample_function_value
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_n_calls (process) result (n_calls)
      integer :: n_calls
      type(process_t), intent(in) :: process
      n_calls = integration_results_get_n_calls (process%results)
    end function process_get_n_calls

    function process_get_integral (process) result (integral)
      real(default) :: integral
      type(process_t), intent(in) :: process
      integral = integration_results_get_integral (process%results)
    end function process_get_integral

    function process_get_error (process) result (error)
      real(default) :: error
      type(process_t), intent(in) :: process
      error = integration_results_get_error (process%results)
    end function process_get_error

    function process_get_accuracy (process) result (accuracy)
      real(default) :: accuracy
      type(process_t), intent(in) :: process
      accuracy = integration_results_get_accuracy (process%results)
    end function process_get_accuracy

    function process_get_chi2 (process) result (chi2)
      real(default) :: chi2
      type(process_t), intent(in) :: process
      chi2 = integration_results_get_chi2 (process%results)
    end function process_get_chi2

    function process_get_time_per_event (process) result (tpe)
      real(default) :: tpe
      type(process_t), intent(in) :: process
      tpe = integration_results_get_time_per_event (process%results)
    end function process_get_time_per_event
```

```
function process_get_efficiency (process) result (efficiency)
  real(default) :: efficiency
  type(process_t), intent(in) :: process
  efficiency = integration_results_get_efficiency (process%results)
end function process_get_efficiency

function process_get_sample_function_value (process) result  (value)
   real(default) :: value
   type(process_t), intent(in) :: process
   value = process%sample_function_value
end function process_get_sample_function_value
```

Return the current (i.e., last) integration pass index, and the index of the last iteration *within* this pass.

⟨*Processes: public*⟩+≡
```
public :: process_get_current_pass
public :: process_get_current_it
```

⟨*Processes: procedures*⟩+≡
```
function process_get_current_pass (process) result (pass)
  integer :: pass
  type(process_t), intent(in) :: process
  pass = integration_results_get_current_pass (process%results)
end function process_get_current_pass

function process_get_current_it (process) result (it)
  integer :: it
  type(process_t), intent(in) :: process
  it = integration_results_get_current_it (process%results)
end function process_get_current_it
```

Return pointers to the sqme and flows evaluators. If no beams are used, these are identical to the evaluators of the hard interaction.

⟨*Processes: public*⟩+≡
```
public :: process_get_eval_sqme_ptr
public :: process_get_eval_flows_ptr
```

⟨*Processes: procedures*⟩+≡
```
function process_get_eval_sqme_ptr (process) result (eval)
  type(evaluator_t), pointer :: eval
  type(process_t), intent(in), target :: process
  if (process%has_extra_evaluators) then
     eval => process%eval_sqme
  else
     eval => hard_interaction_get_eval_sqme_ptr (process%hi)
  end if
end function process_get_eval_sqme_ptr

function process_get_eval_flows_ptr (process) result (eval)
  type(evaluator_t), pointer :: eval
  type(process_t), intent(in), target :: process
  if (process%has_extra_evaluators) then
     eval => process%eval_flows
```

```
      else
         eval => hard_interaction_get_eval_flows_ptr (process%hi)
      end if
    end function process_get_eval_flows_ptr
```

Return pointers to the interaction and to the sqme and flows evaluators of the
hard interaction.

⟨*Processes: public*⟩+≡
```
    public :: process_get_hi_int_ptr
    public :: process_get_hi_eval_sqme_ptr
    public :: process_get_hi_eval_flows_ptr
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_hi_int_ptr (process) result (int)
      type(interaction_t), pointer :: int
      type(process_t), intent(in), target :: process
      int => hard_interaction_get_int_ptr (process%hi)
    end function process_get_hi_int_ptr

    function process_get_hi_eval_sqme_ptr (process) result (eval)
      type(evaluator_t), pointer :: eval
      type(process_t), intent(in), target :: process
      eval => hard_interaction_get_eval_sqme_ptr (process%hi)
    end function process_get_hi_eval_sqme_ptr

    function process_get_hi_eval_flows_ptr (process) result (eval)
      type(evaluator_t), pointer :: eval
      type(process_t), intent(in), target :: process
      eval => hard_interaction_get_eval_flows_ptr (process%hi)
    end function process_get_hi_eval_flows_ptr
```

13.3.6   Setting values directly

Some values can be set directly; this is used when reading an event from file.

⟨*Processes: public*⟩+≡
```
    public :: process_set_scale
    public :: process_set_alpha_s
    public :: process_set_sqme
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_set_scale (process, scale)
      type(process_t), intent(inout) :: process
      real(default), intent(in) :: scale
      process%scale = scale
    end subroutine process_set_scale

    subroutine process_set_alpha_s (process, alpha_s)
      type(process_t), intent(inout) :: process
      real(default), intent(in) :: alpha_s
      process%alpha_s_at_scale = alpha_s
    end subroutine process_set_alpha_s

    subroutine process_set_sqme (process, sqme)
```

```
    type(process_t), intent(inout) :: process
    real(default), intent(in) :: sqme
    process%sqme = sqme
  end subroutine process_set_sqme
```

13.3.7 Process preparation: beams and structure functions

Set up the chain of structure functions. The individual types of structure functions need specific instances. These are just wrappers around the corresponding `strfun_chain` procedures.

If `use_beams` is false, only `sqrts` and `flv` is set, using the decaying particle mass (decay) or the `sqrts` value in the argument list (scattering) to set up a local beam record.

⟨*Processes: public*⟩+≡
```
    public :: process_setup_beams
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_setup_beams &
        (process, beam_data, n_strfun, n_mapping, sqrts, flv)
      type(process_t), intent(inout), target :: process
      type(beam_data_t), intent(in) :: beam_data
      integer, intent(in) :: n_strfun, n_mapping
      real(default), intent(in), optional :: sqrts
      type(flavor_t), dimension(:), intent(in), optional :: flv
      if (.not. process_has_matrix_element (process))  return
      if (process%use_beams) then
         process%beam_data = beam_data
         process%sqrts = beam_data%sqrts
         process%sqrts_known = .true.
         process%n_strfun = n_strfun
         process%azimuthal_dependence = &
              .not. all (polarization_is_diagonal (beam_data%pol))
         process%lab_is_cm_frame = beam_data%lab_is_cm_frame .and. n_strfun == 0
         call strfun_chain_init (process%sfchain, beam_data, n_strfun, n_mapping)
      else
         select case (process%type)
         case (PRC_DECAY)
            process%sqrts = process%mass_in(1)
            call beam_data_init_decay (process%beam_data, process%flv_in)
         case (PRC_SCATTERING)
            if (present (sqrts)) then
               process%sqrts = sqrts
               call beam_data_init_sqrts &
                    (process%beam_data, process%sqrts, process%flv_in)
            else
               call msg_fatal ("Process setup: neither beams nor sqrts are known")
               process%sqrts = 0
            end if
         end select
         process%sqrts_known = .true.
      end if
```

```
      end subroutine process_setup_beams
```

Set the beam momenta directly without changing anything else. This is a short-cut that is needed for initiating cascade decays (i.e., the single beam is the decaying particle).

⟨*Processes: public*⟩+≡
```
    public :: process_set_beam_momenta
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_set_beam_momenta (process, p)
      type(process_t), intent(inout), target :: process
      type(vector4_t), dimension(:), intent(in) :: p
      type(interaction_t), pointer :: hi_int
      if (.not. process_has_matrix_element (process))  return
      if (process%use_beams) then
         call strfun_chain_set_beam_momenta (process%sfchain, p)
      else
         hi_int => hard_interaction_get_int_ptr (process%hi)
         call interaction_set_momenta (hi_int, p, outgoing=.false.)
      end if
      process%sqrts_hat = process%sqrts
      process%lab_is_cm_frame = .false.
      process%beams_are_set = .true.
    end subroutine process_set_beam_momenta
```

Configure structure functions. EPA: support only a single data set.

The index i is the overall structure function counter. line indicates the beam(s) for which the structure function applies, either 1 or 2, or 0 for both beams.

⟨*Processes: public*⟩+≡
```
    public :: process_set_strfun
```

⟨*Processes: interfaces*⟩≡
```
    interface process_set_strfun
       module procedure process_set_strfun_lhapdf
       module procedure process_set_strfun_isr
       module procedure process_set_strfun_epa
       module procedure process_set_strfun_ewa
    end interface
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_set_strfun_lhapdf &
         (process, i, line, lhapdf_data, n_parameters)
      type(process_t), intent(inout), target :: process
      integer, intent(in) :: i, line, n_parameters
      type(lhapdf_data_t), intent(in) :: lhapdf_data
      if (process%use_beams) then
         call strfun_chain_set_strfun &
              (process%sfchain, i, line, lhapdf_data, n_parameters)
      end if
    end subroutine process_set_strfun_lhapdf

    subroutine process_set_strfun_isr &
```

1043

```
         (process, i, line, isr_data, n_parameters)
       type(process_t), intent(inout), target :: process
       integer, intent(in) :: i, line, n_parameters
       type(isr_data_t), intent(in) :: isr_data
       if (process%use_beams) then
          call strfun_chain_set_strfun &
               (process%sfchain, i, line, isr_data, n_parameters)
       end if
     end subroutine process_set_strfun_isr

     subroutine process_set_strfun_epa &
          (process, i, line, epa_data, n_parameters)
       type(process_t), intent(inout), target :: process
       integer, intent(in) :: i, line, n_parameters
 !      type(epa_data_t), dimension(:), intent(in) :: epa_data
       type(epa_data_t), intent(in) :: epa_data
       if (process%use_beams) then
          call strfun_chain_set_strfun &
               (process%sfchain, i, line, epa_data, n_parameters)
       end if
     end subroutine process_set_strfun_epa

     subroutine process_set_strfun_ewa &
          (process, i, line, ewa_data, n_parameters)
       type(process_t), intent(inout), target :: process
       integer, intent(in) :: i, line, n_parameters
 !      type(ewa_data_t), dimension(:), intent(in) :: ewa_data
       type(ewa_data_t), intent(inout) :: ewa_data
       integer :: k
       integer, dimension(process_get_n_tot(process), &
                          process_get_n_flv(process)) :: flvs_tot
       integer, dimension(process_get_n_flv(process)) :: flvs
       flvs_tot = hard_interaction_get_flv_states (process%hi)
       flvs(:) = abs(flvs_tot (line,:))
       do k = 1, size (flvs)
         if (flvs(1) /= flvs (k)) &
           call msg_fatal ("EWA approximation is not applicable when " &
                    // "mixing W and Z for a single beam.")
       end do
       if (flvs(1) < 23 .or. flvs(1) > 24) &
            call msg_fatal ("Hard scattering process does not match EWA.")
       if (process%use_beams) then
          call strfun_chain_set_strfun &
               (process%sfchain, i, line, ewa_data, n_parameters, flvs(1))
       end if
     end subroutine process_set_strfun_ewa
```

Configure structure function mappings.

⟨*Processes: public*⟩+≡

```
    public :: process_set_strfun_mapping
```

⟨*Processes: procedures*⟩+≡

```
    subroutine process_set_strfun_mapping (process, i, index, type, par)
      type(process_t), intent(inout) :: process
```

```
      integer, intent(in) :: i
      integer, intent(in) :: type
      integer, dimension(:), intent(in) :: index
      real(default), dimension(:), intent(in) :: par
      if (process%use_beams) then
         call strfun_chain_set_mapping (process%sfchain, i, index, type, par)
      end if
   end subroutine process_set_strfun_mapping
```

Complete structure function initialization. Make evaluators within the structure
function chain, and the trace evaluator within the hard interaction. Connect the
two, and make another trace evaluator which sums over all quantum numbers.
This evaluator should have only a single matrix element.

⟨*Processes: public*⟩+≡
```
   public :: process_connect_strfun
```

⟨*Processes: procedures*⟩+≡
```
   subroutine process_connect_strfun (process, ok)
     type(process_t), intent(inout), target :: process
     logical, intent(out), optional :: ok
     integer, dimension(:), allocatable :: coll_index
     type(quantum_numbers_mask_t), dimension(:), allocatable :: mask_in
     type(quantum_numbers_mask_t) :: mask_tr
     type(evaluator_t), pointer :: eval_sfchain, eval_hi
     type(interaction_t), pointer :: int_beam, int_hi
     integer :: n_in, i
     if (.not. process_has_matrix_element (process))  return
     n_in  = hard_interaction_get_n_in  (process%hi)
     allocate (mask_in (n_in))
     if (process%use_beams) then
        call strfun_chain_make_evaluators (process%sfchain, ok)
        allocate (coll_index (n_in))
        coll_index = strfun_chain_get_colliding_particles (process%sfchain)
        mask_in = strfun_chain_get_colliding_particles_mask (process%sfchain)
     else
        mask_in = new_quantum_numbers_mask (.true., .true., .true.)
     end if
     call hard_interaction_init_trace &
          (process%hi, mask_in, process%use_hi_color_factors)
     if (process%use_beams) then
        int_beam => strfun_chain_get_beam_int_ptr (process%sfchain)
        eval_sfchain => strfun_chain_get_last_evaluator_ptr (process%sfchain)
        eval_hi => hard_interaction_get_eval_trace_ptr (process%hi)
        int_hi => hard_interaction_get_int_ptr (process%hi)
        mask_tr = new_quantum_numbers_mask (.true., .true., .true.)
        if (associated (eval_sfchain)) then
           do i = 1, n_in
              call interaction_set_source_link &
                   (int_hi, i, eval_sfchain, coll_index(i))
              call evaluator_set_source_link &
                   (eval_hi, i, eval_sfchain, coll_index(i))
           end do
           call evaluator_init_product &
                (process%eval_trace, eval_sfchain, eval_hi, mask_tr, mask_tr)
```

1045

```
            if (evaluator_is_empty (process%eval_trace)) then
               call msg_fatal ("Mismatch between structure functions and hard process")
               if (present (ok))  ok = .false.
               return
            end if
         else
            do i = 1, n_in
               call interaction_set_source_link &
                     (int_hi, i, int_beam, coll_index(i))
               call evaluator_set_source_link &
                     (eval_hi, i, int_beam, coll_index(i))
            end do
            call evaluator_init_product &
                  (process%eval_trace, int_beam, eval_hi, mask_tr, mask_tr)
            if (evaluator_is_empty (process%eval_trace)) then
               call msg_fatal ("Mismatch between beams and hard process")
               if (present (ok))  ok = .false.
               return
            end if
         end if
         process%n_par_strfun = &
               strfun_chain_get_n_parameters_tot (process%sfchain)
         allocate (process%x_strfun (process%n_par_strfun))
      end if
      process%n_par = process%n_par_strfun + process%n_par_hi
      if (present (ok))  ok = .true.
   end subroutine process_connect_strfun
```

Check whether the current setting of relevant variables matches the current
beam setup.

⟨*Processes: public*⟩+≡
```
    public :: process_check_beam_setup
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_check_beam_setup (process, var_list)
      type(process_t), intent(in) :: process
      type(var_list_t), intent(in) :: var_list
      real(default) :: sqrts
      sqrts = var_list_get_rval (var_list, var_str ("sqrts"))
      if (process%use_beams) then
         select case (process%type)
         case (PRC_SCATTERING)
            call beam_data_check_scattering (process%beam_data, sqrts)
         end select
      end if
    end subroutine process_check_beam_setup
```

13.3.8 Process preparation: phase space

Initialize the phase-space forest. First check whether the process exists in
`filename_in`, if present, and try to read it there. If this fails, generate a new

phase-space forest and write it to `filename_out`, if present, otherwise to a temporary file. Then read again.

Because `variable_limits` depends on the structure function setup, structure functions should be done first.

⟨*Processes: public*⟩+≡
```
    public :: process_setup_phase_space
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_setup_phase_space (process, rebuild_phs, &
        phs_par, mapping_defaults, filename_out, filename_in, ok)
      type(process_t), intent(inout), target :: process
      logical, intent(in) :: rebuild_phs
      type(phs_parameters_t), intent(inout) :: phs_par
      type(mapping_defaults_t), intent(in) :: mapping_defaults
      type(string_t), intent(in), optional :: filename_out, filename_in
      logical, intent(out), optional :: ok
      type(string_t) :: filename
      logical :: exist, check
      integer :: extra_off_shell
      type(cascade_set_t) :: cascade_set
      logical :: variable_limits
      integer :: n_in, n_out, n_tot, n_flv
      type(flavor_t), dimension(:,:), allocatable :: flv
      integer :: n_par_strfun
      logical, dimension(:), allocatable :: strfun_rigid
      character(32) :: md5sum_process, md5sum_model, md5sum_parameters
      integer :: unit
      logical :: phs_ok, phs_match, wrote_file
      phs_ok = .false.
      phs_match = .false.
      variable_limits = process%n_strfun /= 0
      n_in  = hard_interaction_get_n_in  (process%hi)
      n_out = hard_interaction_get_n_out (process%hi)
      n_tot = hard_interaction_get_n_tot (process%hi)
      n_flv = hard_interaction_get_n_flv (process%hi)
      allocate (flv (n_tot, n_flv))
      call flavor_init (flv, &
          hard_interaction_get_flv_states (process%hi), process%model)
      md5sum_process = process%md5sum
      md5sum_model = model_get_md5sum (process%model)
      md5sum_parameters = model_get_parameters_md5sum (process%model)
      phs_par%sqrts = process%sqrts
      if (present (filename_in)) then
         filename = filename_in
         call msg_message ("Reading phase-space configuration from file '" &
             // char (filename) // "'")
         check = .false.
      else if (.not. rebuild_phs .and. present (filename_out)) then
         filename = filename_out
         check = .true.
      else
         filename = ""
      end if
      if (filename /= "") then
```

```
inquire (file=char(filename), exist=exist)
if (exist) then
   if (check) then
      call phs_forest_read (process%forest, filename, &
           process%id, n_in, n_out, process%model, phs_ok, &
           md5sum_process, md5sum_model, md5sum_parameters, phs_par, &
           phs_match)
   else
      call phs_forest_read (process%forest, filename, &
           process%id, n_in, n_out, process%model, phs_ok)
      phs_match = .true.
   end if
   if (phs_match) &
        call msg_message ("Read phase-space configuration from file '" &
        // char (filename) // "'...")
   unit = free_unit ()
   open (unit = unit, file = char (filename), action = "read", &
         status = "old")
   process%md5sum_phs = md5sum (unit)
   close (unit)
   if (.not. phs_ok) then
      call msg_fatal ("Phase space file '" // char (filename) &
           // "': No valid phase space for process '" &
           // char (process%id) // "'")
      if (present (ok))  ok = .false.
      return
   end if
   else
      call msg_message ("Phase space file '" // char (filename) &
           // "' not found.")
      phs_match = .false.
   end if
end if
wrote_file = .false.
if (.not. phs_match) then
   call msg_message ("Generating phase space configuration ...")
   LOOP_OFF_SHELL: do extra_off_shell = 0, max (n_tot - 3, 0)
      call cascade_set_generate (cascade_set, &
           process%model, n_in, n_out, flv, phs_par, process%fatal_beam_decay)
      if (cascade_set_is_valid (cascade_set)) then
         exit LOOP_OFF_SHELL
      else if (phs_par%off_shell >= max (n_tot - 3, 0)) then
         call msg_error ("Process '" // char (process%id) &
              // "': no valid phase-space channels found")
         if (present (ok))  ok = .false.
         call cascade_set_final (cascade_set)
         return
      else
         write (msg_buffer, "(A,1x,I0)") &
              "Process '" // char (process%id) &
              // "': no valid phase-space channels found for " &
              // "phs_off_shell =", phs_par%off_shell
         call msg_warning ()
         call msg_message ("Increasing phs_off_shell")
```

```
           phs_par%off_shell = phs_par%off_shell + 1
        end if
     end do LOOP_OFF_SHELL
     unit = free_unit ()
     if (present (filename_out)) then
        open (unit, file=char(filename_out), &
              action="readwrite", status="replace")
     else
        open (unit, action="readwrite", status="scratch")
     end if
     write (unit, *) "process ", char (process%id)
     write (unit, *) "  md5sum_process    = ", '"', md5sum_process, '"'
     write (unit, *) "  md5sum_model      = ", '"', md5sum_model, '"'
     write (unit, *) "  md5sum_parameters = ", '"', md5sum_parameters, '"'
     call phs_parameters_write (phs_par, unit)
     call cascade_set_write_file_format (cascade_set, unit)
     call cascade_set_final (cascade_set)
     rewind (unit)
     call phs_forest_read (process%forest, unit, &
          process%id, n_in, n_out, process%model, phs_ok)
     rewind (unit)
     process%md5sum_phs = md5sum (unit)
     close (unit)
     wrote_file = present (filename_out)
     if (.not. phs_ok) then
        call msg_bug ("Generated phase space file: " &
             // "No valid phase space for process '" &
             // char (process%id) // "'")
     end if
  end if
  call phs_forest_set_flavors (process%forest, flv(:,1))
  call phs_forest_set_parameters &
       (process%forest, mapping_defaults, variable_limits)
  call phs_forest_set_equivalences (process%forest)
  if (process%use_beams) then
     n_par_strfun = strfun_chain_get_n_parameters_tot (process%sfchain)
     allocate (strfun_rigid (n_par_strfun))
     strfun_rigid = strfun_chain_dimension_is_rigid (process%sfchain)
  else
     n_par_strfun = 0
     allocate (strfun_rigid (0))
  end if
  call phs_forest_setup_vamp_equivalences (process%forest, &
       n_par_strfun, strfun_rigid, &
       process%azimuthal_dependence, &
       process%vamp_eq)
  process%n_channels = phs_forest_get_n_channels (process%forest)
  process%n_par_hi = phs_forest_get_n_parameters (process%forest)
  process%n_par = process%n_par_strfun + process%n_par_hi
  allocate (process%x_hi (process%n_par_hi))
  process%x_hi = 0
  allocate (process%x (process%n_par, process%n_channels))
  process%x = 0
  allocate (process%phs_factor (process%n_channels))
```

```
      process%phs_factor = 0
      allocate (process%active_channel (process%n_channels))
      process%active_channel = .true.
      write (msg_buffer, "(A,I0,A,I0,A)")  "... found ", process%n_channels, &
           " phase space channels, collected in ", &
           phs_forest_get_n_groves (process%forest), &
           " groves."
      call msg_message ()
      write (msg_buffer, "(A,I0,A)")  "Phase space: found ", &
           phs_forest_get_n_equivalences (process%forest), &
           " equivalences between channels."
      call msg_message ()
      if (wrote_file) &
           call msg_message ("Wrote phase-space configuration file '" &
           // char (filename_out) // "'.")
      if (present (ok))  ok = .true.
    end subroutine process_setup_phase_space
```

13.3.9   Process preparation: cuts, weight and scale

Compile the cut expression and store it as an evaluation tree inside the process
object.

⟨*Processes: public*⟩+≡
```
    public :: process_setup_cuts
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_setup_cuts (process, parse_node)
      type(process_t), intent(inout), target :: process
      type(parse_node_t), intent(in), target :: parse_node
      call eval_tree_init_lexpr &
           (process%cut_expr, parse_node, process%var_list, process%prt_list)
    end subroutine process_setup_cuts
```

Compile the weight expression and store it as an evaluation tree inside the
process object.

⟨*Processes: public*⟩+≡
```
    public :: process_setup_weight
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_setup_weight (process, parse_node)
      type(process_t), intent(inout), target :: process
      type(parse_node_t), intent(in), target :: parse_node
      call eval_tree_init_expr &
           (process%reweighting_expr, parse_node, process%var_list, &
            process%prt_list)
    end subroutine process_setup_weight
```

Compile the scale expression and store it as an evaluation tree inside the process
object.

⟨*Processes: public*⟩+≡
```
    public :: process_setup_scale
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_setup_scale (process, parse_node)
      type(process_t), intent(inout), target :: process
      type(parse_node_t), intent(in), target :: parse_node
      call eval_tree_init_expr &
           (process%scale_expr, parse_node, process%var_list, process%prt_list)
    end subroutine process_setup_scale
```

13.3.10   Process preparation: VAMP grids

Grid parameters: transparent container.

⟨*Processes: public*⟩+≡
```
    public :: grid_parameters_t
```

⟨*Processes: types*⟩+≡
```
    type :: grid_parameters_t
       integer :: threshold_calls = 0
       integer :: min_calls_per_channel = 10
       integer :: min_calls_per_bin = 10
       integer :: min_bins = 3
       integer :: max_bins = 20
       logical :: stratified = .true.
       logical :: use_vamp_equivalences = .true.
       real(default) :: channel_weights_power = 0.25_default
    end type grid_parameters_t
```

I/O:

⟨*Processes: procedures*⟩+≡
```
    subroutine grid_parameters_write (grid_par, unit)
      type(grid_parameters_t), intent(in) :: grid_par
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, *) "threshold_calls       = ", grid_par%threshold_calls
      write (u, *) "min_calls_per_channel = ", grid_par%min_calls_per_channel
      write (u, *) "min_calls_per_bin     = ", grid_par%min_calls_per_bin
      write (u, *) "min_bins              = ", grid_par%min_bins
      write (u, *) "max_bins              = ", grid_par%max_bins
      write (u, *) "stratified            = ", grid_par%stratified
      write (u, *) "use_vamp_equivalences = ", grid_par%use_vamp_equivalences
      write (u, *) "channel_weights_power = ", grid_par%channel_weights_power
    end subroutine grid_parameters_write
```

⟨*Processes: procedures*⟩+≡
```
    subroutine grid_parameters_read (grid_par, unit)
      type(grid_parameters_t), intent(out) :: grid_par
      integer, intent(in) :: unit
      character(30) :: dummy
      character :: equals
      read (unit, *) dummy, equals, grid_par%threshold_calls
      read (unit, *) dummy, equals, grid_par%min_calls_per_channel
      read (unit, *) dummy, equals, grid_par%min_calls_per_bin
```

```
      read (unit, *) dummy, equals, grid_par%min_bins
      read (unit, *) dummy, equals, grid_par%max_bins
      read (unit, *) dummy, equals, grid_par%stratified
      read (unit, *) dummy, equals, grid_par%use_vamp_equivalences
      read (unit, *) dummy, equals, grid_par%channel_weights_power
    end subroutine grid_parameters_read
```

⟨*Processes: interfaces*⟩+≡
```
    interface operator(==)
       module procedure grid_parameters_eq
    end interface
```
⟨*Processes: procedures*⟩+≡
```
    function grid_parameters_eq (gp1, gp2) result (eq)
      logical :: eq
      type(grid_parameters_t), intent(in) :: gp1, gp2
      eq = gp1%threshold_calls == gp2%threshold_calls &
           .and. gp1%min_calls_per_channel   == gp2%min_calls_per_channel &
           .and. gp1%min_calls_per_bin       == gp2%min_calls_per_bin     &
           .and. gp1%min_bins                == gp2%min_bins              &
           .and. gp1%max_bins                == gp2%max_bins              &
           .and.(gp1%stratified         .eqv. gp2%stratified            )&
           .and.(gp1%use_vamp_equivalences .eqv. gp2%use_vamp_equivalences)&
           .and. gp1%channel_weights_power   == gp2%channel_weights_power
    end function grid_parameters_eq
```

⟨*Processes: interfaces*⟩+≡
```
    interface operator(/=)
       module procedure grid_parameters_ne
    end interface
```
⟨*Processes: procedures*⟩+≡
```
    function grid_parameters_ne (gp1, gp2) result (ne)
      logical :: ne
      type(grid_parameters_t), intent(in) :: gp1, gp2
      ne = gp1%threshold_calls /= gp2%threshold_calls &
           .or. gp1%min_calls_per_channel   /= gp2%min_calls_per_channel &
           .or. gp1%min_calls_per_bin       /= gp2%min_calls_per_bin     &
           .or. gp1%min_bins                /= gp2%min_bins              &
           .or. gp1%max_bins                /= gp2%max_bins              &
           .or.(gp1%stratified         .neqv. gp2%stratified            )&
           .or.(gp1%use_vamp_equivalences .neqv. gp2%use_vamp_equivalences)&
           .or. gp1%channel_weights_power   /= gp2%channel_weights_power
    end function grid_parameters_ne
```

Initialize the grids with uniform channel weight.

⟨*Processes: public*⟩+≡
```
    public :: process_setup_grids
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_setup_grids (process, grid_parameters, calls)
      type(process_t), intent(inout), target :: process
      type(grid_parameters_t), intent(in) :: grid_parameters
      integer, intent(in) :: calls
```

```
      integer, dimension(:), allocatable :: num_div
      real(default), dimension(:), allocatable :: weights
      real(default), dimension(:,:), allocatable :: region
      integer :: min_calls
      allocate (num_div (process%n_par))
      min_calls = grid_parameters%min_calls_per_bin * process%n_channels
      if (min_calls /= 0) then
         process%n_bins =  max (grid_parameters%min_bins, &
               min (calls / min_calls, grid_parameters%max_bins))
      else
         process%n_bins = grid_parameters%max_bins
      end if
      allocate (region (2, process%n_par))
      region(1,:) = 0
      region(2,:) = 1
      allocate (weights (process%n_channels))
      weights = 1
      num_div = process%n_bins
      call msg_message ("Creating VAMP integration grids:")
      if (grid_parameters%use_vamp_equivalences) &
           call msg_message ("Using phase-space channel equivalences.")
      call vamp_create_grids (process%grids, region, calls, weights, &
           num_div=num_div, stratified=grid_parameters%stratified)
      process%vamp_grids_defined = .true.
    end subroutine process_setup_grids
```

13.3.11   Process preparation: Helicity selection counters

The helicity selection counters can be activated and reset at startup, removing
unnecessary helicities after `cutoff` tries.

⟨*Processes: public*⟩+≡
```
    public :: process_reset_helicity_selection
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_reset_helicity_selection (process, threshold, cutoff)
      type(process_t), intent(inout) :: process
      real(default), intent(in) :: threshold
      integer, intent(in) :: cutoff
      call hard_interaction_reset_helicity_selection &
           (process%hi, threshold, cutoff)
    end subroutine process_reset_helicity_selection
```

13.3.12   Matrix element evaluation

Kinematics. This evaluates structure functions as far as momenta are concerned.
The evaluator links automatically transfer the incoming momenta to the hard
interaction. All phase space factors are evaluated, and the resulting momenta
are stored back in the hard interaction, from there transferred to the appropriate
evaluators.

Once the particle momenta are known, they are transferred to the `prt_list` that is used for cut/weight/scale evaluation. The energy scale is computed right here.

If `ok` is false, there is no valid momentum assignement for the given x parameters, and the event must be dropped.

⟨*Processes: procedures*⟩+≡

```
subroutine process_set_kinematics (process, x_in, channel, ok)
  type(process_t), intent(inout), target :: process
  real(default), dimension(:), intent(in) :: x_in
  integer, intent(in) :: channel
  logical, intent(out) :: ok
  type(interaction_t), pointer :: int
  type(evaluator_t), pointer :: eval
  integer :: i
  real(default) :: lda
  process%x_hi = x_in(:process%n_par_hi)
  process%x_strfun = x_in(process%n_par_hi+1:)
!    process%x_strfun = x_in(:process%n_par_strfun)
!    process%x_hi = x_in(process%n_par_strfun+1:)
  process%channel = channel
  int => hard_interaction_get_int_ptr (process%hi)
  eval => hard_interaction_get_eval_trace_ptr (process%hi)
  if (process%use_beams) then
     call strfun_chain_set_kinematics (process%sfchain, process%x_strfun)
     call interaction_receive_momenta (int)
     process%beams_are_set = .true.
     process%sqrts_hat = sqrt (max (interaction_get_s (int), 0._default))
  else if (.not. process%beams_are_set) then
     select case (process%type)
     case (PRC_DECAY)
        call interaction_set_momenta (int, &
             (/ vector4_at_rest (process%mass_in(1)) /), &
             outgoing=.false.)
     case (PRC_SCATTERING)
        call interaction_set_momenta (int, &
             colliding_momenta (process%sqrts, process%mass_in), &
             outgoing=.false.)
     end select
     process%sqrts_hat = process%sqrts
  end if
  select case (process%type)
  case (PRC_DECAY)
     process%flux_factor = &
          twopi4 / (2 * process%mass_in(1))
  case (PRC_SCATTERING)
     lda = lambda (process%sqrts_hat ** 2, &
                           process%mass_in(1) ** 2, &
                           process%mass_in(2) ** 2)
     if (lda <= 0) then
        ok = .false.; return
     end if
     process%flux_factor = &
          conv * twopi4 / (2 * sqrt (lda))
```

```
      end select
      process%sqrts_hat_known = .true.
      if (.not. process%lab_is_cm_frame) then
         process%lt_cm_to_lab = interaction_get_cm_transformation (int)
         call phs_forest_set_prt_in (process%forest, int, process%lt_cm_to_lab)
      else
         call phs_forest_set_prt_in (process%forest, int)
      end if
      process%x(:process%n_par_hi,channel) = process%x_hi
      forall (i = 1 : process%n_par_strfun)
         process%x(process%n_par_hi+i,:) = process%x_strfun(i)
      end forall
      call phs_forest_evaluate_phase_space (process%forest, &
           channel, process%active_channel, process%sqrts_hat, &
           process%x, process%phs_factor, process%phs_volume, ok)
      if (ok) then
         if (process%lab_is_cm_frame) then
            call phs_forest_get_prt_out (process%forest, int)
         else
            call phs_forest_get_prt_out &
                 (process%forest, int, process%lt_cm_to_lab)
         end if
         call interaction_momenta_to_prt_list (int, process%prt_list)
         call process_compute_scale (process)
      end if
      call evaluator_receive_momenta (eval)
      if (process%use_beams) &
           call evaluator_receive_momenta (process%eval_trace)
    end subroutine process_set_kinematics
```

Evaluate the (jacobian) factor associated to the VAMP grids and the phase-space factor array for the given integration channel.

⟨*Processes: procedures*⟩+≡

```
    subroutine process_compute_vamp_phs_factor (process, weights)
      type(process_t), intent(inout), target :: process
      real(default), dimension(:), intent(in) :: weights
      real(default), dimension(process%n_channels) :: vamp_prob
      real(default) :: dp
      integer :: i
      do i = 1, process%n_channels
         if (process%active_channel(i)) then
            vamp_prob(i) = &
                 vamp_probability (process%grids%grids(i), process%x(:,i))
         else
            vamp_prob(i) = 0
         end if
      end do
      dp = dot_product (weights, vamp_prob / process%phs_factor)
      if (dp /= 0) then
         process%vamp_phs_factor = vamp_prob(process%channel) / dp
      else
         process%vamp_phs_factor = 0
      end if
```

```
    end subroutine process_compute_vamp_phs_factor
```

Evaluate the cut expression and return a boolean flag (true means to continue evaluation, false to drop the event).

⟨*Processes: procedures*⟩+≡
```
    function process_passes_cuts (process) result (flag)
      logical :: flag
      type(process_t), intent(inout), target :: process
      if (eval_tree_is_defined (process%cut_expr)) then
         call eval_tree_evaluate (process%cut_expr)
         if (eval_tree_result_is_known (process%cut_expr)) then
            flag = eval_tree_get_log (process%cut_expr)
         else
            flag = .true.
         end if
      else
         flag = .true.
      end if
    end function process_passes_cuts
```

Evaluate the weight expression and set the value.

⟨*Processes: procedures*⟩+≡
```
    subroutine process_compute_reweighting_factor (process)
      type(process_t), intent(inout), target :: process
      if (eval_tree_is_defined (process%reweighting_expr)) then
         call eval_tree_evaluate (process%reweighting_expr)
         if (eval_tree_result_is_known (process%reweighting_expr)) then
            process%reweighting_factor = &
                 eval_tree_get_real (process%reweighting_expr)
         else
            process%reweighting_factor = 1
         end if
      else
         process%reweighting_factor = 1
      end if
    end subroutine process_compute_reweighting_factor
```

Evaluate the scale expression and return a real value. If the scale expression is undefined, return the c.m. energy of the hard interaction.

⟨*Processes: procedures*⟩+≡
```
    subroutine process_compute_scale (process)
      type(process_t), intent(inout), target :: process
      if (eval_tree_is_defined (process%scale_expr)) then
         call eval_tree_evaluate (process%scale_expr)
         if (eval_tree_result_is_known (process%scale_expr)) then
            process%scale = eval_tree_get_real (process%scale_expr)
         else
            process%scale = process%sqrts_hat
         end if
      else
         process%scale = process%sqrts_hat
      end if
```

```
          end subroutine process_compute_scale
```

Update the α_s value used by the matrix element code, depending on the computed scale.

⟨*Processes: interfaces*⟩+≡
```
  interface
     double precision function alphasPDF (Q)
        double precision, intent(in) :: Q
     end function alphasPDF
  end interface
```

⟨*Processes: procedures*⟩+≡
```
     subroutine process_update_alpha_s (process)
       type(process_t), intent(inout) :: process
       real(default) :: scale, nf, as_mz, mz, lambda, alpha_s
       integer :: order
       scale = process%scale
       as_mz = process%alpha_s_mz
       mz = process%mz
       lambda = process%lambda_qcd
       order = process%alpha_s_order
       nf = process%alpha_s_nf
       if (.not. process%alpha_s_is_fixed) then
          if (process%alpha_s_from_lhapdf) then
             alpha_s = alphasPDF (dble (scale))
          else
             if (process%alpha_s_from_mz) then
                if (process%alpha_s_mz_is_known) then
                   if (process%mz_is_known) then
                      alpha_s = running_as (scale, &
                           al_mz=as_mz, mz=mz, order=order, nf=nf)
                   else
                      alpha_s = running_as (scale, &
                           al_mz=as_mz, order=order, nf=nf)
                   end if
                else
                   if (process%mz_is_known) then
                      alpha_s = running_as (scale, &
                           mz=mz, order=order, nf=nf)
                   else
                      alpha_s = running_as (scale, &
                           order=order, nf=nf)
                   end if
                end if
             else
                alpha_s = running_as_lam (nf, scale, lambda, order=order)
             end if
          end if
          process%alpha_s_at_scale = alpha_s
          call hard_interaction_update_alpha_s (process%hi, alpha_s)
       end if
     end subroutine process_update_alpha_s
```

1057

Evaluate the structure function values, the hard matrix element, and the follow-up evaluators. We obtain the squared matrix element value for the current event.

⟨*Processes: procedures*⟩+≡
```
subroutine process_evaluate (process)
  type(process_t), intent(inout), target :: process
  if (process%use_beams) then
     call strfun_chain_evaluate (process%sfchain, process%scale)
     process%sf_mapping_factor = &
           strfun_chain_get_mapping_factor (process%sfchain)
  else
     process%sf_mapping_factor = 1
  end if
  call hard_interaction_evaluate (process%hi)
  if (process%has_extra_evaluators) then
     call evaluator_evaluate (process%eval_trace)
     process%sqme = evaluator_sum (process%eval_trace)
  else
     process%sqme = evaluator_sum &
           (hard_interaction_get_eval_trace_ptr (process%hi)) &
           * process%averaging_factor
  end if
!    call process_write (process, 66); stop
end subroutine process_evaluate
```

Return the squared matrix element of the hard interaction for the given momenta, traced over all quantum numbers. This is independent of beam setup, structure functions, phase space etc.

⟨*Processes: procedures*⟩+≡
```
function process_compute_sqme_sum (process, p) result (sqme)
  real(default) :: sqme
  type(process_t), intent(inout), target :: process
  type(vector4_t), dimension(:), intent(in) :: p
  sqme = hard_interaction_compute_sqme_sum (process%hi, p)
end function process_compute_sqme_sum
```

13.3.13   Access VAMP data

Compute the reweighting efficiency for the current grids, suitable averaged over all active channels.

⟨*Processes: procedures*⟩+≡
```
function process_get_vamp_efficiency_array (process) result (efficiency)
  real(default), dimension(:), allocatable :: efficiency
  type(process_t), intent(in) :: process
  allocate (efficiency (process%n_channels))
  where (process%grids%grids%f_max /= 0)
     efficiency = process%grids%grids%mu(1) / abs (process%grids%grids%f_max)
  elsewhere
     efficiency = 0
  end where
end function process_get_vamp_efficiency_array
```

```
function process_get_vamp_efficiency (process) result (efficiency)
  real(default) :: efficiency
  type(process_t), intent(in) :: process
  real(default), dimension(:), allocatable :: weight
  real(default) :: norm
  allocate (weight (process%n_channels))
  weight = process%grids%weights * abs (process%grids%grids%f_max)
  norm = sum (weight)
  if (norm /= 0) then
     efficiency = &
          dot_product (process_get_vamp_efficiency_array (process), weight) &
          / norm
  else
     efficiency = 1
  end if
end function process_get_vamp_efficiency
```

13.3.14 Integration

This executes one or more iterations of the VAMP integration routine. The flags
determine whether to discard previous results, to adapt grids before integration,
and to adapt the relative channel weights. The final result is entered into the
results record.

⟨*Processes: public*⟩+≡
```
    public :: process_integrate
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_integrate (process, rng, &
        grid_parameters, pass, it1, it2, calls, &
        discard_integrals, adapt_grids, adapt_weights, print_current, &
        time_estimate, &
        grids_filename, md5sum_beams, md5sum_sf_list, md5sum_mappings, &
        md5sum_cuts, md5sum_weight, md5sum_scale)
      type(process_t), intent(inout), target :: process
      type(tao_random_state), intent(inout) :: rng
      type(grid_parameters_t), intent(in) :: grid_parameters
      integer, intent(in) :: pass, it1, it2, calls
      logical, intent(in) :: discard_integrals
      logical, intent(in) :: adapt_grids
      logical, intent(in) :: adapt_weights
      logical, intent(in) :: print_current
      logical, intent(in) :: time_estimate
      type(string_t), intent(in), optional :: grids_filename
      character(32), intent(in), optional :: &
          md5sum_beams, md5sum_sf_list, md5sum_mappings
      character(32), intent(in), optional :: &
          md5sum_cuts, md5sum_weight, md5sum_scale
      integer :: it
      real(default) :: integral, error, efficiency
      type(time_t) :: time_start, time_end
      character(32) :: md5sum_process, md5sum_model, md5sum_parameters
      character(32) :: md5sum_phs
```

```fortran
real(default) :: sqrts
real(default), dimension(:), allocatable :: grove_weight
integer :: u
if (it1 > it2)  return
u = logfile_unit ()
md5sum_process = process%md5sum
md5sum_model = model_get_md5sum (process%model)
md5sum_parameters = model_get_parameters_md5sum (process%model)
md5sum_phs = process%md5sum_phs
sqrts = process%sqrts
if (discard_integrals .and. it1==1) then
   if (grid_parameters%use_vamp_equivalences) then
      call vamp_discard_integrals (process%grids, &
           calls, stratified=grid_parameters%stratified, eq=process%vamp_eq)
   else
      call vamp_discard_integrals (process%grids, &
           calls, stratified=grid_parameters%stratified)
   end if
end if
process%beams_are_set = .false.
do it = it1, it2
   if (adapt_grids) then
      call process_adapt_grids (process)
   end if
   if (adapt_weights) then
      call process_adapt_channel_weights (process, grid_parameters, calls)
   end if
   if (time_estimate)  time_start = time_current ()
   if (grid_parameters%use_vamp_equivalences) then
      call vamp_sample_grids &
           (rng, process%grids, sample_function, process%store_index, 1, &
            eq=process%vamp_eq, &
            history=process%v_history(it:), &
            histories=process%v_histories(it:,:), &
            integral=integral, std_dev=error)
   else
      call vamp_sample_grids &
           (rng, process%grids, sample_function, process%store_index, 1, &
            history=process%v_history(it:), &
            histories=process%v_histories(it:,:), &
            integral=integral, std_dev=error)
   end if
   if (time_estimate)  time_end = time_current ()
   efficiency = process_get_vamp_efficiency (process)
   call process_get_grove_weights (process, grove_weight)
   if (time_estimate) then
      call integration_results_append (process%results, &
           process%type, pass, 1, calls, &
           integral, error, efficiency, grove_weight, time_start, time_end)
   else
      call integration_results_append (process%results, &
           process%type, pass, 1, calls, &
           integral, error, efficiency, grove_weight)
   end if
```

1060

```
             if (present (grids_filename)) then
                call write_grid_file (grids_filename, process%id, &
                     md5sum_process, md5sum_model, md5sum_parameters, md5sum_phs, &
                     md5sum_beams, md5sum_sf_list, md5sum_mappings, &
                     md5sum_cuts, md5sum_weight, md5sum_scale, &
                     grid_parameters, process%results, process%grids)
             end if
             if (print_current) then
                call integration_results_write_current (process%results)
                call integration_results_write_current (process%results, unit=u)
                flush (u)
             end if
          end do
       end subroutine process_integrate
```

This should be executed instead if the process has no matrix element.

⟨*Processes: public*⟩+≡
```
    public :: process_do_dummy_integration
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_do_dummy_integration (process)
      type(process_t), intent(inout) :: process
      call integration_results_append (process%results, &
           process%type, 1, 1, 0, &
           0._default, 0._default, 0._default)
    end subroutine process_do_dummy_integration
```

Create the VAMP history information:

⟨*Processes: public*⟩+≡
```
    public :: process_init_vamp_history
    public :: process_final_vamp_history
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_init_vamp_history (process, n_iterations)
      type(process_t), intent(inout) :: process
      integer, intent(in) :: n_iterations
      call process_final_vamp_history (process)
      allocate (process%v_history (n_iterations))
      allocate (process%v_histories &
           (n_iterations, process_get_n_channels (process)))
      call vamp_create_history (process%v_history, verbose=.false.)
      call vamp_create_history (process%v_histories, verbose=.false.)
    end subroutine process_init_vamp_history

    subroutine process_final_vamp_history (process)
      type(process_t), intent(inout) :: process
      if (allocated (process%v_history)) then
         call vamp_delete_history (process%v_history)
         deallocate (process%v_history)
      end if
      if (allocated (process%v_histories)) then
         call vamp_delete_history (process%v_histories)
         deallocate (process%v_histories)
      end if
```

```
      end subroutine process_final_vamp_history
```

Display the time estimate on screen

⟨*Processes: public*⟩+≡
```
    public :: process_write_time_estimate
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_write_time_estimate (process, unit)
      type(process_t), intent(in) :: process
      integer, intent(in), optional :: unit
      real(default) :: time_per_event, time_per_10k
      character(80) :: time_string
      type(time_t) :: t
      time_per_event = integration_results_get_time_per_event (process%results)
      time_per_10k = 10000 * time_per_event
      t = time_from_seconds (time_per_10k)
      if (time_per_10k == 0) then
         write (time_string, "(A)")  "0"
      else if (time_per_10k < 100) then
         call time_write (t, time_string, milliseconds=.true.)
      else if (time_per_10k < 86400) then
         call time_write (t, time_string)
      else if (time_per_10k < 3600) then
         call time_write (t, time_string, seconds=.false.)
      else
         call time_write (t, time_string, days=.true., seconds=.false.)
      end if
      write (msg_buffer, "(A)")  "Process '" // char (process%id) // "': "
      call msg_message ()
      write (msg_buffer, "(A)")  "  time estimate for 10000 unweighted events = " &
           // trim (adjustl (time_string))
      call msg_message (unit=unit)
      call write_hline (unit)
    end subroutine process_write_time_estimate
```

Write the VAMP grid file including a header containing metadata.

⟨*Processes: procedures*⟩+≡
```
    subroutine write_grid_file (filename, process_id, &
         md5sum_process, md5sum_model, md5sum_parameters, md5sum_phs, &
         md5sum_beams, md5sum_sf_list, md5sum_mappings, &
         md5sum_cuts, md5sum_weight, md5sum_scale, &
         grid_parameters, results, grids)
      type(string_t), intent(in) :: filename, process_id
      character(32), intent(in) :: md5sum_process, md5sum_model
      character(32), intent(in) :: md5sum_phs, md5sum_parameters
      character(32), intent(in) :: md5sum_beams, md5sum_sf_list, md5sum_mappings
      character(32), intent(in) :: md5sum_cuts, md5sum_weight, md5sum_scale
      type(grid_parameters_t), intent(in) :: grid_parameters
      type(integration_results_t), intent(in) :: results
      type(vamp_grids), intent(in) :: grids
      integer :: u
      u = free_unit ()
      open (file = char (filename), unit = u, &
```

```
        action = "write", status = "replace")
    write (u, *) "process ", char (process_id)
    write (u, *) "  md5sum_process     = ", '"', md5sum_process, '"'
    write (u, *) "  md5sum_model       = ", '"', md5sum_model, '"'
    write (u, *) "  md5sum_parameters  = ", '"', md5sum_parameters, '"'
    write (u, *) "  md5sum_phase_space = ", '"', md5sum_phs, '"'
    write (u, *) "  md5sum_beams       = ", '"', md5sum_beams, '"'
    write (u, *) "  md5sum_sf_list     = ", '"', md5sum_sf_list, '"'
    write (u, *) "  md5sum_mappings    = ", '"', md5sum_mappings, '"'
    write (u, *) "  md5sum_cuts        = ", '"', md5sum_cuts, '"'
    write (u, *) "  md5sum_weight      = ", '"', md5sum_weight, '"'
    write (u, *) "  md5sum_scale       = ", '"', md5sum_scale, '"'
    write (u, *)
    call grid_parameters_write (grid_parameters, u)
    write (u, *)
    call integration_results_write &
         (results, u, verbose = .true.)
    write (u, *)
    call vamp_write_grids (grids, u, write_integrals = .true.)
    close (u)
  end subroutine write_grid_file
```

Attempt to read the VAMP grid file, checking metadata for consistency. Also read the integration results as far as they are known.

⟨*Processes: procedures*⟩+≡

```
  subroutine read_grid_file (filename, process_id, &
       md5sum_process, md5sum_model, md5sum_parameters, md5sum_phs, &
       md5sum_beams, md5sum_sf_list, md5sum_mappings, &
       md5sum_cuts, md5sum_weight, md5sum_scale, &
       grid_parameters, results, grids, &
       pass, n_calls, ok)
    type(string_t), intent(in) :: filename, process_id
    character(32), intent(in) :: md5sum_process, md5sum_model
    character(32), intent(in) :: md5sum_phs, md5sum_parameters
    character(32), intent(in) :: md5sum_beams, md5sum_sf_list, md5sum_mappings
    character(32), intent(in) :: md5sum_cuts, md5sum_weight, md5sum_scale
    type(grid_parameters_t), intent(in) :: grid_parameters
    type(integration_results_t), intent(out) :: results
    type(vamp_grids), intent(inout) :: grids
    integer, dimension(:), intent(in) :: pass, n_calls
    logical, intent(out) :: ok
    integer :: u
    logical :: exist
    character(80) :: buffer
    character :: equals
    character(32) :: md5sum
    type(grid_parameters_t) :: grid_parameters_file
    type(integration_results_t) :: results_file
    ok = .false.
    inquire (file = char (filename), exist = exist)
    if (.not. exist)  return
    call msg_message ("Reading integration grids and results from file '" &
         // char (filename) // "':")
```

```
u = free_unit ()
open (file = char (filename), unit = u, action = "read", status = "old")
read (u, *)  buffer
if (trim (adjustl (buffer)) /= "process") then
   call msg_fatal ("Grid file: missing 'process' tag")
   close (u);  return
end if
read (u, *)  buffer, equals, md5sum
if (md5sum /= md5sum_process) then
   call msg_message &
        ("Process configuration has changed, discarding old grid file")
   close (u);  return
end if
read (u, *)  buffer, equals, md5sum
if (md5sum /= md5sum_model) then
   call msg_message &
        ("Model has changed, discarding old grid file")
   close (u);  return
end if
read (u, *)  buffer, equals, md5sum
if (md5sum /= md5sum_parameters) then
   call msg_message &
        ("Model parameters have changed, discarding old grid file")
   close (u);  return
end if
read (u, *)  buffer, equals, md5sum
if (md5sum /= md5sum_phs) then
   call msg_message &
        ("Phase-space setup has changed, discarding old grid file")
   close (u);  return
end if
read (u, *)  buffer, equals, md5sum
if (md5sum /= md5sum_beams) then
   call msg_message &
        ("Beam setup has changed, discarding old grid file")
   close (u);  return
end if
read (u, *)  buffer, equals, md5sum
if (md5sum /= md5sum_sf_list) then
   call msg_message &
        ("Structure-function setup has changed, discarding old grid file")
   close (u);  return
end if
read (u, *)  buffer, equals, md5sum
if (md5sum /= md5sum_mappings) then
   call msg_message &
        ("Mapping scale parameters have changed, discarding old grid file")
   close (u);  return
end if
read (u, *)  buffer, equals, md5sum
if (md5sum /= md5sum_cuts) then
   call msg_message &
        ("Cut configuration has changed, discarding old grid file")
   close (u);  return
```

```
         end if
         read (u, *)  buffer, equals, md5sum
         if (md5sum /= md5sum_weight) then
            call msg_message &
                 ("Weight expression has changed, discarding old grid file")
            close (u);  return
         end if
         read (u, *)  buffer, equals, md5sum
         if (md5sum /= md5sum_scale) then
            call msg_message &
                 ("Scale expression have changed, discarding old grid file")
            close (u);  return
         end if
         read (u, *)
         call grid_parameters_read (grid_parameters_file, u)
         if (grid_parameters_file /= grid_parameters) then
            call msg_message &
                 ("Grid parameters have changed, discarding old grid file")
            close (u);  return
         end if
         read (u, *)
         call integration_results_read (results_file, u)
         if (.not. integration_results_iterations_are_consistent &
              (results_file, pass, n_calls)) then
            call msg_message &
                 ("Iteration parameters have changed, discarding old grid file")
            close (u);  return
         end if
         results = results_file
         read (u, *)
         call vamp_read_grids (grids, u)
         close (u)
         ok = .true.
      end subroutine read_grid_file
```

Wrapper for grid file reading.

⟨*Processes: public*⟩+≡
```
      public :: process_read_grid_file
```

⟨*Processes: procedures*⟩+≡
```
      subroutine process_read_grid_file (process, filename, &
           md5sum_beams, md5sum_sf_list, md5sum_mappings, &
           md5sum_cuts, md5sum_weight, md5sum_scale,&
           grid_parameters, pass, n_calls, ok)
        type(process_t), intent(inout) :: process
        type(string_t), intent(in) :: filename
        character(32), intent(in) :: md5sum_beams, md5sum_sf_list, md5sum_mappings
        character(32), intent(in) :: md5sum_cuts, md5sum_weight, md5sum_scale
        type(grid_parameters_t), intent(in) :: grid_parameters
        integer, dimension(:), intent(in) :: pass, n_calls
        logical, intent(out) :: ok
        character(32) :: md5sum_process, md5sum_model
        character(32) :: md5sum_parameters, md5sum_phs
        md5sum_process = process%md5sum
```

```
      md5sum_model = model_get_md5sum (process%model)
      md5sum_parameters = model_get_parameters_md5sum (process%model)
      md5sum_phs = process%md5sum_phs
      call read_grid_file (filename, process%id, &
           md5sum_process, md5sum_model, md5sum_parameters, md5sum_phs, &
           md5sum_beams, md5sum_sf_list, md5sum_mappings, &
           md5sum_cuts, md5sum_weight, md5sum_scale, &
           grid_parameters, process%results, process%grids, &
           pass, n_calls, ok)
    end subroutine process_read_grid_file
```

Adapt the binning of the VAMP grids. This is just a wrapper.

⟨*Processes: procedures*⟩+≡
```
    subroutine process_adapt_grids (process)
      type(process_t), intent(inout), target :: process
      call vamp_refine_grids (process%grids)
    end subroutine process_adapt_grids
```

Refine the channel weights. We use a power weight just as for the individual bins. The results are averaged within each grove. Then, we check if the resulting weights would lead to a too small number of calls within any channel, which is corrected. The result is fed into VAMP.

⟨*Processes: procedures*⟩+≡
```
    subroutine process_adapt_channel_weights (process, grid_parameters, calls)
      type(process_t), intent(inout), target :: process
      type(grid_parameters_t), intent(in) :: grid_parameters
      integer, intent(in) :: calls
      real(default), dimension(:), allocatable :: weights
      integer :: g, i0, i1, n
      real(default) :: sum_weights, weight_min
      logical, dimension(:), allocatable :: weight_underflow
      real(default) :: sum_weight_underflow
      integer :: n_underflow
      allocate (weights (process%n_channels))
      weights = process%grids%weights &
           * vamp_get_variance (process%grids%grids) &
           ** grid_parameters%channel_weights_power
      do g = 1, phs_forest_get_n_groves (process%forest)
         call phs_forest_get_grove_bounds (process%forest, g, i0, i1, n)
         weights(i0:i1) = sum (weights(i0:i1)) / n
      end do
      sum_weights = sum (weights)
      if (sum_weights /= 0) then
         weights = weights / sum (weights)
         if (grid_parameters%threshold_calls /= 0) then
            weight_min = &
                 real (grid_parameters%threshold_calls, default) &
                 / calls
            weight_underflow = weights /= 0 .and. weights < weight_min
            n_underflow = count (weight_underflow)
            sum_weight_underflow = sum (weights, mask=weight_underflow)
            where (weight_underflow)
               weights = weight_min
```

```
          elsewhere
             weights = weights &
                   * (1 - n_underflow * weight_min) / (1 - sum_weight_underflow)
          end where
       end if
       call vamp_update_weights (process%grids, weights)
   end if
 end subroutine process_adapt_channel_weights
```

Return a concise table of channel weights: sum over all channels that contribute
to a grove and return the grove weights.

⟨Processes: procedures⟩+≡
```
 subroutine process_get_grove_weights (process, grove_weight)
   type(process_t), intent(in) :: process
   real(default), dimension(:), allocatable, intent(out) :: grove_weight
   integer :: n_groves, g, i0, i1, n
   n_groves = phs_forest_get_n_groves (process%forest)
   allocate (grove_weight (n_groves))
   do g = 1, n_groves
      call phs_forest_get_grove_bounds (process%forest, g, i0, i1, n)
      grove_weight(g) = sum (process%grids%weights(i0:i1))
   end do
 end subroutine process_get_grove_weights
```

13.3.15   Event generation

Initialize event generation. For the process setup, this means that the evaluators
for the exclusive matrix element with and without color-flow decomposition is
activated.

For the hard-interaction evaluators, we select only those entries which are
supported by the beam/structure function setup. E.g., we select diagonal helic-
ity only, or sum over helicity if the beams are unpolarized. When constructing
the process evaluators, we multiply the beams by the hard-interaction evalua-
tors and trace over all incoming-particle quantum numbers (except for color in
the color-flow evaluator).

⟨Processes: public⟩+≡
```
 public :: process_setup_event_generation
```

⟨Processes: procedures⟩+≡
```
 subroutine process_setup_event_generation (process, qn_mask_in)
   type(process_t), intent(inout), target :: process
   type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_in
   integer, dimension(:), allocatable :: coll_index
   type(quantum_numbers_mask_t), dimension(:), allocatable :: mask_in
   type(quantum_numbers_mask_t) :: mask_conn_sqme, mask_conn_flows
   type(evaluator_t), pointer :: eval_sfchain, eval_sqme, eval_flows
   type(interaction_t), pointer :: int_hi, int_beam
   integer :: n_in, n_out, n_tot, i
   type(evaluator_t), target :: eval_con
   if (.not. process_has_matrix_element (process)) then
      call msg_warning ("Process '" // char (process%id) // "': " &
            // "matrix element vanishes, no events can be generated")
```

```
          return
       end if
       call hard_interaction_final_sqme (process%hi)
       call hard_interaction_final_flows (process%hi)
       call evaluator_final (process%eval_beam_flows)
       call evaluator_final (process%eval_sqme)
       call evaluator_final (process%eval_flows)
       n_in  = hard_interaction_get_n_in  (process%hi)
       n_out = hard_interaction_get_n_out (process%hi)
       n_tot = hard_interaction_get_n_tot (process%hi)
       int_hi => hard_interaction_get_int_ptr (process%hi)
       call interaction_reset_momenta (int_hi)
       allocate (mask_in (n_in))
       if (process%use_beams) then
          allocate (coll_index (n_in))
          coll_index = strfun_chain_get_colliding_particles (process%sfchain)
          mask_in = strfun_chain_get_colliding_particles_mask (process%sfchain)
          mask_conn_sqme = new_quantum_numbers_mask (.false., .true., .true.)
          mask_conn_flows = new_quantum_numbers_mask (.false., .false., .true.)
       else if (present (qn_mask_in)) then
          mask_in = qn_mask_in
       else
          mask_in = new_quantum_numbers_mask (.false., .false., .true.)
       end if
       call hard_interaction_init_sqme  (process%hi, mask_in, &
            process%use_hi_color_factors)
       call hard_interaction_init_flows (process%hi, mask_in)
       int_beam     => strfun_chain_get_beam_int_ptr         (process%sfchain)
       eval_sfchain => strfun_chain_get_last_evaluator_ptr (process%sfchain)
       eval_sqme    => hard_interaction_get_eval_sqme_ptr  (process%hi)
       eval_flows   => hard_interaction_get_eval_flows_ptr (process%hi)
!      print *, "Hard interaction"
!      call interaction_write (int_hi)
!      print *, "Evaluator: HI: SQME"
!      call evaluator_write (eval_sqme)
!      print *, "Evaluator: HI: flows"
!      call evaluator_write (eval_flows)
       if (process%use_beams) then
          if (associated (eval_sfchain)) then
             call evaluator_init_color_contractions &
                  (process%eval_beam_flows, evaluator_get_int_ptr (eval_sfchain))
             do i = 1, n_in
                call evaluator_set_source_link &
                     (eval_sqme,  i, eval_sfchain, coll_index(i))
                call evaluator_set_source_link &
                     (eval_flows, i, process%eval_beam_flows, coll_index(i))
             end do
             if (process%has_extra_evaluators) then
                call evaluator_init_product (process%eval_sqme, &
                     eval_sfchain, eval_sqme,  mask_conn_sqme)
                call evaluator_init_product (process%eval_flows, &
                     process%eval_beam_flows, eval_flows, mask_conn_flows)
             end if
          else
```

```
         call evaluator_init_color_contractions &
              (process%eval_beam_flows, int_beam)
         do i = 1, n_in
            call evaluator_set_source_link &
                 (eval_sqme,  i, int_beam, coll_index(i))
            call evaluator_set_source_link &
                 (eval_flows, i, process%eval_beam_flows, coll_index(i))
         end do
         if (process%has_extra_evaluators) then
            call evaluator_init_product (process%eval_sqme, &
                 int_beam, eval_sqme, mask_conn_sqme)
            call evaluator_init_product (process%eval_flows, &
                 process%eval_beam_flows, eval_flows, mask_conn_flows)
         end if
      end if
!        print *, "Evaluator: Beams+HI: SQME"
!        call evaluator_write (process%eval_sqme)
!        print *, "Evaluator: Beams+HI: flows"
!        call evaluator_write (process%eval_flows)
!        call process_write (process, 77)
   end if
 end subroutine process_setup_event_generation
```

Generate a weighted event. We have to select a channel. The output is the
event weight, unmodified. The `sample_function` fully constructs the event in
the `process` object, so the output x array is not needed.

Analysis is not yet implemented; we need a means to pass the event weight
to the recording functions.

⟨*Processes: public*⟩+≡
```
   public :: process_generate_weighted_event
```

⟨*Processes: procedures*⟩+≡
```
   subroutine process_generate_weighted_event (process, rng, weight)
     type(process_t), intent(inout), target :: process
     type(tao_random_state), intent(inout) :: rng
     real(default), intent(out) :: weight
     real(default), dimension(process%n_par) :: x
     call vamp_next_event &
          (x, rng, process%grids, &
           sample_function, process%store_index, phi_trivial, &
           weight=weight)
     call process_complete_evaluators (process)
   end subroutine process_generate_weighted_event
```

Generate an unweighted event. Rejection is done by `VAMP`. The optional `excess`
is nonzero by the excess weight if an event weight exceeds the precalculated
maximum that is used for rejection. After the event has been generated, it can
be analyzed.

The transformation function `phi` is trivial but has to be supplied.

⟨*Processes: public*⟩+≡
```
   public :: process_generate_unweighted_event
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_generate_unweighted_event (process, rng, excess)
  type(process_t), intent(inout), target :: process
  type(tao_random_state), intent(inout) :: rng
  real(default), intent(out), optional :: excess
  real(default), dimension(process%n_par) :: x
  call vamp_next_event &
       (x, rng, process%grids, &
        sample_function, process%store_index, phi_trivial, &
        excess=excess)
  call process_complete_evaluators (process)
end subroutine process_generate_unweighted_event

function phi_trivial (xi, channel_dummy) result (x)
  real(default), dimension(:), intent(in) :: xi
  integer, intent(in) :: channel_dummy
  real(default), dimension(size(xi)) :: x
  x = xi
end function phi_trivial
```

Complete the event: compute amplitudes/probabilities for exclusive quantum numbers.

⟨*Processes: procedures*⟩+≡
```
subroutine process_complete_evaluators (process)
  type(process_t), intent(inout), target :: process
  if (process%use_beams) then
     call evaluator_receive_momenta (process%eval_beam_flows)
     call evaluator_evaluate (process%eval_beam_flows)
  end if
  call hard_interaction_evaluate_sqme (process%hi)
  call hard_interaction_evaluate_flows (process%hi)
  if (process%has_extra_evaluators) then
     call evaluator_receive_momenta (process%eval_sqme)
     call evaluator_receive_momenta (process%eval_flows)
     call evaluator_evaluate (process%eval_sqme)
     call evaluator_evaluate (process%eval_flows)
  end if
end subroutine process_complete_evaluators
```

This procedure is used for checking whether some of the final-state particles can initiate decay cascades.

⟨*Processes: public*⟩+≡
```
public :: process_get_unstable_products
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_get_unstable_products (process, flv_unstable)
  type(process_t), intent(in) :: process
  type(flavor_t), dimension(:), intent(out), allocatable :: flv_unstable
  call hard_interaction_get_unstable_products (process%hi, flv_unstable)
end subroutine process_get_unstable_products
```

When the event is generated externally (e.g., read from file), we need to fill the particle list with the process record in order to analyze it. This is done here:

⟨*Processes: public*⟩+≡
```
    public :: process_set_particles
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_set_particles (process, particle_set)
      type(process_t), intent(inout) :: process
      type(particle_set_t), intent(in) :: particle_set
      call particle_set_to_prt_list (particle_set, process%prt_list)
    end subroutine process_set_particles
```

13.3.16   Results output

⟨*Processes: public*⟩+≡
```
    public :: process_results_write_header
    public :: process_results_write_entry
    public :: process_results_write_current
    public :: process_results_write_average
    public :: process_results_write_current_average
    public :: process_results_write_footer
    public :: process_results_write
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_results_write_header (process, unit, logfile)
      type(process_t), intent(in) :: process
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: logfile
      call write_dline (unit)
      call write_header (process%type, unit, logfile)
      call write_dline (unit)
    end subroutine process_results_write_header

    subroutine process_results_write_entry (process, it, unit)
      type(process_t), intent(in) :: process
      integer, intent(in) :: it
      integer, intent(in), optional :: unit
      call integration_results_write_entry (process%results, it, unit)
    end subroutine process_results_write_entry

    subroutine process_results_write_current (process, unit)
      type(process_t), intent(in) :: process
      integer, intent(in), optional :: unit
      call integration_results_write_current (process%results, unit)
    end subroutine process_results_write_current

    subroutine process_results_write_average (process, pass, unit)
      type(process_t), intent(in) :: process
      integer, intent(in) :: pass
      integer, intent(in), optional :: unit
      call write_hline (unit)
      call integration_results_write_average (process%results, pass, unit)
      call write_hline (unit)
    end subroutine process_results_write_average

    subroutine process_results_write_current_average (process, unit)
```

```
      type(process_t), intent(in) :: process
      integer, intent(in), optional :: unit
      call write_hline (unit)
      call integration_results_write_current_average (process%results, unit)
      call write_hline (unit)
    end subroutine process_results_write_current_average

    subroutine process_results_write_footer (process, unit, no_line)
      type(process_t), intent(in) :: process
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: no_line
      if (present (no_line)) then
         if (.not. no_line)  call write_dline (unit)
      else
         call write_dline (unit)
      end if
      call integration_results_write_current_average (process%results, unit)
      call write_dline (unit)
    end subroutine process_results_write_footer

    subroutine process_results_write (process, unit)
      type(process_t), intent(in) :: process
      integer, intent(in), optional :: unit
      call integration_results_write (process%results, unit)
    end subroutine process_results_write
```

Record the integration results in the process library entry.

⟨*Processes: public*⟩+≡
```
    public :: process_record_integral
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_record_integral (process, var_list)
      type(process_t), intent(inout) :: process
      type(var_list_t), intent(inout) :: var_list
      integer :: n_calls
      real(default) :: integral, error, accuracy, chi2, efficiency
      n_calls = integration_results_get_n_calls (process%results)
      integral = integration_results_get_integral (process%results)
      error = integration_results_get_error (process%results)
      accuracy = integration_results_get_accuracy (process%results)
      chi2 = integration_results_get_chi2 (process%results)
      efficiency = integration_results_get_efficiency (process%results)
      call process_library_record_integral (process%prc_lib, process%id, &
           n_calls, integral, error, accuracy, chi2, efficiency)
      call var_list_init_process_results (var_list, process%id, &
           n_calls, integral, error, accuracy, chi2, efficiency)
    end subroutine process_record_integral
```

13.3.17   Copies

Process copies are used for decay chains (and such). We deep-copy most components, in particular the hard-interaction and decay-forest workspaces. Read-only components (variable list) and eval trees are transferred as shallow copies.

We do not copy the extra evaluators that convolute beams and hard matrix elements. Actually, beams should not be defined for a cascade decay process in the first place, but we do not enforce this. We also skip the integration results.

Furthermore, we have to reassign all external links between interactions within the process copy to point to the copy, not the original.

The copy is linked to the original by a pointer. Its initial state is `in_use`.

⟨*Processes: procedures*⟩+≡
```
subroutine process_make_copy (process, original)
  type(process_t), intent(inout), target :: process
  type(process_t), intent(in), target :: original
  type(process_t), pointer :: copy
  type(interaction_t), pointer :: beam_int, copy_beam_int
  type(interaction_t), pointer :: hi_int, copy_hi_int
  type(evaluator_t), pointer :: hi_eval_trace, copy_hi_eval_trace
  type(evaluator_t), pointer :: hi_eval_sqme, copy_hi_eval_sqme
  type(evaluator_t), pointer :: hi_eval_flows, copy_hi_eval_flows
  allocate (copy)
  copy%type = original%type
  copy%is_original = .false.
  copy%original => original
  copy%initialized = original%initialized
  copy%has_matrix_element = original%has_matrix_element
  copy%use_hi_color_factors = original%use_hi_color_factors
  copy%use_beams = original%use_beams
  copy%has_extra_evaluators = .false.
  copy%id = original%id
  copy%prc_lib => original%prc_lib
  copy%lib_index = original%lib_index
  copy%store_index = original%store_index
  copy%model => original%model
  if (original%use_beams) then
     copy%n_strfun = original%n_strfun
     copy%n_par_strfun = original%n_par_strfun
  end if
  copy%n_par_hi = original%n_par_hi
  copy%n_par = original%n_par
  copy%azimuthal_dependence = original%azimuthal_dependence
  copy%vamp_grids_defined = original%vamp_grids_defined
  copy%sqrts_known = original%sqrts_known
  copy%sqrts = original%sqrts
  if (original%use_beams) then
     if (allocated (original%x_strfun)) &
         allocate (copy%x_strfun (size (original%x_strfun)))
  end if
  if (allocated (original%x_hi)) &
      allocate (copy%x_hi (size (original%x_hi)))
  copy%n_channels = original%n_channels
  if (allocated (original%x)) &
      allocate (copy%x (size (original%x, 1), size (original%x, 2)))
  if (allocated (original%phs_factor)) &
      allocate (copy%phs_factor (size (original%phs_factor)))
  if (allocated (original%mass_in)) then
     allocate (copy%mass_in (size (original%mass_in)))
```

1073

```
      copy%mass_in = original%mass_in
end if
copy%averaging_factor = original%averaging_factor
if (original%use_beams)  copy%sfchain = original%sfchain
copy%hi = original%hi
if (original%use_beams)  copy%eval_trace = original%eval_trace
!  copy%eval_beam_flows = original%eval_beam_flows
!  copy%eval_sqme = original%eval_sqme
!  copy%eval_flows = original%eval_flows
copy%forest = original%forest
copy%vamp_eq = original%vamp_eq
copy%prt_list = original%prt_list
copy%var_list = original%var_list
copy%cut_expr = original%cut_expr
copy%reweighting_expr = original%reweighting_expr
copy%scale_expr = original%scale_expr
if (allocated (original%active_channel)) then
   allocate (copy%active_channel (size (original%active_channel)))
   copy%active_channel = original%active_channel
end if
call vamp_copy_grids (copy%grids, original%grids)
beam_int => strfun_chain_get_beam_int_ptr (original%sfchain)
hi_int => hard_interaction_get_int_ptr (original%hi)
hi_eval_trace => hard_interaction_get_eval_trace_ptr (original%hi)
hi_eval_sqme => hard_interaction_get_eval_sqme_ptr (original%hi)
hi_eval_flows => hard_interaction_get_eval_flows_ptr (original%hi)
copy_beam_int => strfun_chain_get_beam_int_ptr (copy%sfchain)
copy_hi_int => hard_interaction_get_int_ptr (copy%hi)
copy_hi_eval_trace => hard_interaction_get_eval_trace_ptr (copy%hi)
copy_hi_eval_sqme => hard_interaction_get_eval_sqme_ptr (copy%hi)
copy_hi_eval_flows => hard_interaction_get_eval_flows_ptr (copy%hi)
select case (original%type)
case (PRC_SCATTERING)
   call msg_bug ("Process copy for scattering processes not implemented")
case (PRC_DECAY)
   call interaction_reassign_links &
        (copy_hi_int, beam_int, copy_beam_int)
   call evaluator_reassign_links &
        (copy_hi_eval_trace, beam_int, copy_beam_int)
   call evaluator_reassign_links &
        (copy_hi_eval_sqme, beam_int, copy_beam_int)
   call evaluator_reassign_links &
        (copy_hi_eval_flows, beam_int, copy_beam_int)
   call evaluator_reassign_links &
        (copy_hi_eval_trace, hi_int, copy_hi_int)
   call evaluator_reassign_links &
        (copy_hi_eval_sqme, hi_int, copy_hi_int)
   call evaluator_reassign_links &
        (copy_hi_eval_flows, hi_int, copy_hi_int)
   call evaluator_reassign_links &
        (copy%eval_trace, beam_int, copy_beam_int)
   call evaluator_reassign_links &
        (copy%eval_sqme, beam_int, copy_beam_int)
   call evaluator_reassign_links &
```

1074

```
                (copy%eval_flows, beam_int, copy_beam_int)
        call evaluator_reassign_links &
                (copy%eval_trace, hi_eval_trace, copy_hi_eval_trace)
        call evaluator_reassign_links &
                (copy%eval_sqme, hi_eval_sqme, copy_hi_eval_sqme)
        call evaluator_reassign_links &
                (copy%eval_flows, hi_eval_flows, copy_hi_eval_flows)
      end select
      process%copy => copy
    end subroutine process_make_copy
```

Request a process copy. If there is a copy currently not in use, activate it. Otherwise, make a new copy. In the original process, point to this copy as the working copy.

⟨*Processes: public*⟩+≡
```
    public :: process_request_copy
```

⟨*Processes: procedures*⟩+≡
```
    recursive subroutine process_request_copy (process, copy, original)
      type(process_t), intent(inout), target :: process
      type(process_t), pointer :: copy
      type(process_t), intent(inout), target, optional :: original
      if (associated (process%copy)) then
        if (process%copy%in_use) then
          if (present (original)) then
            call process_request_copy (process%copy, copy, original)
          else
            call process_request_copy (process%copy, copy, process)
          end if
        else
          copy => process%copy
          copy%in_use = .true.
          if (present (original)) then
            original%working_copy => copy
          else
            process%working_copy => copy
          end if
        end if
      else
        if (present (original)) then
          call process_make_copy (process, original)
        else
          call process_make_copy (process, original=process)
        end if
        copy => process%copy
        copy%in_use = .true.
        if (present (original)) then
          original%working_copy => copy
        else
          process%working_copy => copy
        end if
      end if
    end subroutine process_request_copy
```

Return the working copy of a process. If there is none, return the process itself.

⟨*Processes: procedures*⟩+≡
```
function process_get_working_copy_ptr (process) result (copy)
  type(process_t), intent(in), target :: process
  type(process_t), pointer :: copy
  if (associated (process%working_copy)) then
     copy => process%working_copy
  else
     copy => process
  end if
end function process_get_working_copy_ptr
```

Mark this copy of the current process as not in use, so it can be requested again. (Use with care! The copy will not reflect changes made to the original process.)

⟨*Processes: public*⟩+≡
```
public :: process_free_copy
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_free_copy (process)
  type(process_t), intent(inout), target :: process
  process%in_use = .false.
  if (associated (process%original)) then
     process%original%working_copy => null ()
  end if
end subroutine process_free_copy
```

Delete all copies.

⟨*Process: public*⟩≡
```
public :: process_delete_copy
```

⟨*Processes: procedures*⟩+≡
```
recursive subroutine process_delete_copies (process)
  type(process_t), intent(inout), target :: process
  if (associated (process%copy)) then
     call process_final (process%copy)
     deallocate (process%copy)
  end if
end subroutine process_delete_copies
```

13.3.18 Process store

The process store is a container for the list of all processes. The list is expanded as needed during program execution. The container is implemented as a module variable. Thus, there is only one process store in the program.

The reason for this is the sampling function, which needs to access it without referencing it as an argument. Instead, it takes an integer argument which identifies the process. For direct access, we maintain a process pointer array as a shortcut to the list.

Type and object

⟨*Processes: types*⟩+≡
```
    type :: process_entry_t
       type(process_t) :: process
       type(process_entry_t), pointer :: next => null ()
    end type process_entry_t
```

⟨*Processes: types*⟩+≡
```
    type :: process_p
       type(process_t), pointer :: ptr => null ()
    end type process_p
```

⟨*Processes: types*⟩+≡
```
    type :: process_store_t
       integer :: n = 0
       type(process_entry_t), pointer :: first => null ()
       type(process_entry_t), pointer :: last => null ()
       type(process_p), dimension(:), allocatable :: proc
    end type process_store_t
```

⟨*Processes: variables*⟩≡
```
    type(process_store_t), save :: store
```

Finalize. Delete the list explicitly, the pointer array is just deallocated.

⟨*Processes: public*⟩+≡
```
    public :: process_store_final
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_store_final ()
      type(process_entry_t), pointer :: current
      if (allocated (store%proc))  deallocate (store%proc)
      store%last => null ()
      do while (associated (store%first))
        current => store%first
        store%first => current%next
        call process_final (current%process)
        deallocate (current)
      end do
      store%n = 0
    end subroutine process_store_final
```

Handlers for unloading and reloading process libraries

⟨*Processes: public*⟩+≡
```
    public :: process_store_unload
    public :: process_store_reload
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_store_unload (libname)
      type(string_t), intent(in) :: libname
      type(process_entry_t), pointer :: entry
      entry => store%first
      do while (associated (entry))
```

1077

```
          if (process_library_get_name (entry%process%prc_lib) == libname) &
             call worker (entry%process)
          entry => entry%next
       end do

    contains

       recursive subroutine worker (process)
         type(process_t), intent(inout), target :: process
         call hard_interaction_unload (process%hi)
         if (associated (process%copy)) call worker (process%copy)
       end subroutine worker

    end subroutine process_store_unload

    subroutine process_store_reload (libname)
       type(string_t), intent(in) :: libname
       type(process_entry_t), pointer :: entry
       entry => store%first
       do while (associated (entry))
          if (process_library_get_name (entry%process%prc_lib) == libname) &
             call worker (entry%process)
          entry => entry%next
       end do

    contains

       recursive subroutine worker (process)
         type(process_t), intent(inout), target :: process
         call hard_interaction_reload (process%hi, process%prc_lib)
         if (associated (process%copy)) call worker (process%copy)
       end subroutine worker

    end subroutine process_store_reload
```

Write all contents. This produces lots of output.

⟨*Processes: public*⟩+≡
```
    public :: process_store_write
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_store_write (unit)
       integer, intent(in), optional :: unit
       type(process_t), pointer :: process
       integer :: u, i
       u = output_unit (unit);  if (u < 0)  return
       write (u, *)  repeat ("%", 78)
       write (u, *)  "Process store contents"
       do i = 1, store%n
          write (u, *)  repeat ("%", 78)
          write (u, *)  "Process No.", i
          process => store%proc(i)%ptr
          call process_write (process, unit)
       end do
       write (u, *)  "Process store end"
```

```
      write (u, *)  repeat ("%", 78)
    end subroutine process_store_write
```

Write integration results (summary)

⟨*Processes: public*⟩+≡
```
    public :: process_store_write_results
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_store_write_results (unit)
      integer, intent(in), optional :: unit
      type(process_t), pointer :: process
      type(string_t), dimension(:), allocatable :: process_id
      real(default), dimension(:), allocatable :: integral, error
      type(string_t), dimension(:), allocatable :: phys_unit
      integer :: u, i, process_id_len
      character(12) :: fmt
      u = output_unit (unit);  if (u < 0)  return
      allocate (process_id (store%n), phys_unit (store%n))
      allocate (integral (store%n), error (store%n))
      do i = 1, store%n
         process => store%proc(i)%ptr
         if (process%initialized) then
            process_id(i) = process%id
            integral(i) = process_get_integral (process)
            error(i) = process_get_error (process)
            select case (process%type)
            case (PRC_DECAY);       phys_unit(i) = "GeV"
            case (PRC_SCATTERING);  phys_unit(i) = "fb"
            case default;           phys_unit(i) = "[undefined]"
            end select
         else
            process_id(i) = ""
         end if
      end do
      write (u, "(A)")  "|========================    Results Summary    ======================
      if (store%n == 0) then
         write (u, *) "[empty]"
      else
         process_id_len = maxval (len (process_id))
         write (fmt, "(A,I0,A)")  "(1x,A", process_id_len + 1, ")"
         do i = 1, store%n
            if (process_id(i) /= "") then
               write (u, fmt, advance="no")  char (process_id(i)) // ":"
               write (u, "(1x, 1PE15.8, 1x, '+-', 1x, 1PE8.2)", advance="no") &
                     integral(i), error(i)
               write (u, "(1x, A)")  char (phys_unit(i))
            end if
         end do
      end if
      write (u, "(A)")  "|==================================================================
    end subroutine process_store_write_results
```

Accessing contents

Return the current number of processes.

⟨*Processes: procedures*⟩+≡
```
function process_store_get_n_processes () result (n)
  integer :: n
  n = store%n
end function process_store_get_n_processes
```

Return a pointer to the process entry with given ID. If it does not exist, return a null pointer.

⟨*Processes: procedures*⟩+≡
```
function process_store_get_entry_ptr (process_id) result (entry)
  type(process_entry_t), pointer :: entry
  type(string_t), intent(in) :: process_id
  entry => store%first
  do while (associated (entry))
     if (entry%process%id == process_id)  exit
     entry => entry%next
  end do
end function process_store_get_entry_ptr
```

Return the index of the process entry with given ID within the process store. If it does not exist, return zero.

⟨*Processes: procedures*⟩+≡
```
function process_store_get_process_index (process_id) result (process_index)
  integer :: process_index
  type(string_t), intent(in) :: process_id
  type(process_entry_t), pointer :: entry
  entry => process_store_get_entry_ptr (process_id)
  if (associated (entry)) then
     process_index = entry%process%store_index
  else
     process_index = 0
  end if
end function process_store_get_process_index
```

Return a pointer to the process with index `i` or alphanumeric ID.

⟨*Processes: public*⟩+≡
```
public :: process_store_get_process_ptr
```

⟨*Processes: interfaces*⟩+≡
```
interface process_store_get_process_ptr
   module procedure process_store_get_process_ptr_int
   module procedure process_store_get_process_ptr_id
end interface
```

⟨*Processes: procedures*⟩+≡
```
function process_store_get_process_ptr_int (i) result (process)
  type(process_t), pointer :: process
  integer, intent(in) :: i
  if (i > 0 .and. i <= size (store%proc)) then
```

```
      process => store%proc(i)%ptr
   else
      process => null ()
   end if
end function process_store_get_process_ptr_int

function process_store_get_process_ptr_id (id) result (process)
  type(process_t), pointer :: process
  type(string_t), intent(in) :: id
  integer :: i
  do i = 1, store%n
     process => store%proc(i)%ptr
     if (process%id == id)  return
  end do
  process => null ()
end function process_store_get_process_ptr_id
```

Filling the process store

Append a new process entry and return a pointer to it, unless the process
already exists. If the process exists, finalize it and return the pointer for fresh
initialization. If it does not exist, allocate a new entry and update the shortcut
array. If the latter is full, expand by a fixed block size.

⟨*Processes: procedures*⟩+≡

```
   function process_store_get_fresh_process_ptr (process_id) result (process)
     type(process_t), pointer :: process
     type(string_t), intent(in) :: process_id
     type(process_entry_t), pointer :: current, entry
     integer :: i
     integer, parameter :: BLOCK_SIZE = 10
     current => process_store_get_entry_ptr (process_id)
     if (associated (current)) then
        call process_final (current%process)
     else
        allocate (current)
        if (store%n == 0) then
           allocate (store%proc (BLOCK_SIZE))
           store%first => current
        else
           store%last%next => current
        end if
        store%last => current
        store%n = store%n + 1
        if (store%n <= size (store%proc)) then
           store%proc(store%n)%ptr => current%process
        else
           deallocate (store%proc)
           allocate (store%proc (store%n + BLOCK_SIZE))
           i = 1
           entry => store%first
           do while (associated (entry))
              store%proc(i)%ptr => entry%process;  i = i + 1
```

1081

```
              entry => entry%next
           end do
        end if
     end if
     process => current%process
  end function process_store_get_fresh_process_ptr
```

Append a new process entry (or find an existing one) and initialize it with a particular hard process, model pointer and total energy. Return a pointer to the process object, so further process preparation can be done by the caller. Allocate or expand the array as needed.

⟨*Processes: public*⟩+≡
```
  public :: process_store_init_process
```

⟨*Processes: procedures*⟩+≡
```
  subroutine process_store_init_process (process, &
       prc_lib, process_id, model, lhapdf_status, var_list, use_beams)
    type(process_t), pointer :: process
    type(process_library_t), intent(inout), target :: prc_lib
    type(string_t), intent(in) :: process_id
    type(model_t), intent(in), target :: model
    type(lhapdf_status_t), intent(inout) :: lhapdf_status
    type(var_list_t), intent(in), optional, target :: var_list
    logical, intent(in), optional :: use_beams
    integer :: process_lib_index, process_store_index
    procedure(prclib_unload_hook), pointer :: unload_hook
    procedure(prclib_reload_hook), pointer :: reload_hook
    process_lib_index = process_library_get_process_index (prc_lib, process_id)
    if (process_lib_index == 0) then
       call msg_fatal ("Process '" // char (process_id) &
            // "' is not available.")
    end if
    process_store_index = process_store_get_process_index (process_id)
    process => process_store_get_fresh_process_ptr (process_id)
    if (process_store_index == 0) then
       process_store_index = process_store_get_n_processes ()
    end if
    unload_hook => process_store_unload
    reload_hook => process_store_reload
    call process_library_set_unload_hook (prc_lib, unload_hook)
    call process_library_set_reload_hook (prc_lib, reload_hook)
    call process_init &
         (process, prc_lib, process_lib_index, process_store_index, &
          process_id, model, lhapdf_status, var_list, use_beams)
  end subroutine process_store_init_process
```

13.3.19   Sampling function

This is the function that computes the squared matrix element in the form needed for VAMP integration. It computes and multiplies the flux factor for the incoming particles, the phase-space factors of the integration channels combined with the VAMP grid jacobians to a common phase-space factor, the phase-space

volume, and the squared matrix element of the hard interaction. The latter is only evaluated if the event passes cuts.

⟨*Processes: procedures*⟩+≡
```
    function sample_function (xi, prc_index, weights, channel, grids) result (f)
      real(default) :: f
      real(default), dimension(:), intent(in) :: xi
      integer, intent(in) :: prc_index
      real(default), dimension(:), intent(in), optional :: weights
      integer, intent(in), optional :: channel
      type(vamp_grid), dimension(:), intent(in), optional :: grids
      type(process_t), pointer :: process
      logical :: ok
      process => process_get_working_copy_ptr (store%proc(prc_index)%ptr)
      call process_set_kinematics (process, xi, channel, ok)
      if (ok)  ok = process_passes_cuts (process)
      if (ok) then
         call process_compute_vamp_phs_factor (process, weights)
         call process_update_alpha_s (process)
         call process_evaluate (process)
         call process_compute_reweighting_factor (process)
         process%sample_function_value = &
              process%flux_factor &
              * process%sf_mapping_factor &
              * process%vamp_phs_factor &
              * process%phs_volume &
              * process%sqme &
              * process%reweighting_factor
      else
         process%sample_function_value = 0
      end if
      f = process%sample_function_value
    end function sample_function
```

13.3.20   Test

⟨*Processes: public*⟩+≡
```
    public :: process_test
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_test ()
      type(os_data_t) :: os_data
      type(process_library_t) :: prc_lib
      type(model_t), pointer :: model
      print *, "*** Load process library"
      call process_library_init (prc_lib, var_str("proc"), os_data)
      call process_library_load (prc_lib, os_data)
      print *
      print *, "*** Read model file"
      call syntax_model_file_init ()
      call model_list_read_model &
           (var_str("QCD"), var_str("test.mdl"), os_data, model)
      call syntax_pexpr_init ()
      call syntax_phs_forest_init ()
```

```
      print *
      call process_test1 (prc_lib, model)
      print *
      call process_test2 (prc_lib, model)
      print *
      call process_test3 (prc_lib, model)
!       print *
!       call process_test4 (prc_lib, model)
      print *
      print *, "* Cleanup"
      call process_store_final ()
      call syntax_pexpr_final ()
      call syntax_phs_forest_final ()
      call syntax_model_file_final ()
      call process_library_final (prc_lib)
    end subroutine process_test
```

Test decay process: $Z \rightarrow e^+e^-$ (colorless); polarized and unpolarized

⟨*Processes: procedures*⟩+≡

```
    subroutine process_test1 (prc_lib, model)
      type(process_library_t), intent(inout) :: prc_lib
      type(model_t), intent(in), target :: model
      type(process_t), pointer :: process
      type(lhapdf_status_t) :: lhapdf_status
      type(phs_parameters_t) :: phs_par
      type(mapping_defaults_t) :: mapping_defaults
      type(flavor_t), dimension(1) :: flv
      type(polarization_t), dimension(1) :: pol
      type(beam_data_t) :: beam_data
      type(grid_parameters_t) :: grid_parameters
      type(tao_random_state) :: rng
      integer :: i
      logical :: rebuild_phs = .true.
      logical :: discard_integrals, adapt_grids, adapt_weights, print_current
      logical :: time_estimate = .true.
      print *, "*** Test decay process"
      print *
      print *, "* Initialization"
      call tao_random_create (rng, 0)
      call process_store_init_process &
           (process, prc_lib, var_str ("zee"), model, lhapdf_status)
      print *, "  Process ID = ", char (process%id)
      print *
      print *, "*** Beam/strfun setup (unpolarized)"
      print *
      call flavor_init (flv, (/ 23 /), model)
      call polarization_init_unpolarized (pol(1), flv(1))
      call beam_data_init_decay (beam_data, flv, pol)
      call process_setup_beams (process, beam_data, 0, 0)
      call process_connect_strfun (process)
      print *
      print *, "* Phase space setup"
      call process_setup_phase_space (process, rebuild_phs, &
```

```
        phs_par, mapping_defaults, filename_out=var_str("zee.phs"))
print *
print *, "*** Test integration"
print *, "* Grids setup"
grid_parameters%stratified = .false.
call process_setup_grids (process, grid_parameters, calls=1000)
print *
print *, "* 1 iteration with minimal number of calls"
call process_results_write_header (process)
do i = 1, 1
   discard_integrals = i==1
   adapt_grids = .true.
   adapt_weights = .true.
   print_current = .true.
   call process_integrate (process, rng, grid_parameters, &
        1, 1, 1, 9, &
        discard_integrals, adapt_grids, adapt_weights, print_current, &
        time_estimate)
end do
call process_results_write_footer (process)
print *
print *, "* Process written to 'fort.60'"
call process_write (process, 60)
print *
print *, "*** Beam/strfun setup (polarized)"
call process_store_init_process &
     (process, prc_lib, var_str ("zee"), model, lhapdf_status)
call flavor_init (flv, (/ 23 /), model)
call polarization_init_axis &
     (pol(1), flv(1), (/ 0._default, 0._default, 1._default/))
call beam_data_init_decay (beam_data, flv, pol)
call process_setup_beams (process, beam_data, 0, 0)
call process_connect_strfun (process)
print *
print *, "* Phase space setup"
call process_setup_phase_space (process, rebuild_phs, &
     phs_par, mapping_defaults, filename_out=var_str("zee.phs"))
print *
print *, "*** Test integration"
print *, "* Grids setup"
grid_parameters%stratified = .false.
call process_setup_grids (process, grid_parameters, calls=1000)
print *
print *, "* 3 + 3 iterations"
call process_results_write_header (process)
do i = 1, 3
   discard_integrals = i==1
   adapt_grids = .true.
   adapt_weights = .true.
   print_current = .true.
   call process_integrate (process, rng, grid_parameters, &
        1, 1, 1, 10000, &
        discard_integrals, adapt_grids, adapt_weights, print_current, &
        time_estimate)
```

```
      end do
      call process_results_write_current_average (process)
      call process_integrate (process, rng, grid_parameters, &
           2, 1, 3, 10000, &
           .true., .true., .false., .true., .true.)
      call process_results_write_footer (process)
      call process_write_time_estimate (process)
      print *
      print *, "* Process written to 'fort.61'"
      call process_write (process, 61)
    end subroutine process_test1
```

Test decay process: $Z \rightarrow qq$ with $q = u, d$.

⟨*Processes: procedures*⟩+≡

```
    subroutine process_test2 (prc_lib, model)
      type(process_library_t), intent(inout) :: prc_lib
      type(model_t), intent(in), target :: model
      type(lhapdf_status_t) :: lhapdf_status
      type(process_t), pointer :: process
      type(phs_parameters_t) :: phs_par
      type(mapping_defaults_t) :: mapping_defaults
      type(flavor_t), dimension(1) :: flv
      type(polarization_t), dimension(1) :: pol
      type(beam_data_t) :: beam_data
      type(grid_parameters_t) :: grid_parameters
      type(tao_random_state) :: rng
      real(default) :: weight
      logical :: time_estimate = .true.
      integer :: i
      logical :: rebuild_phs = .true.
      logical :: discard_integrals, adapt_grids, adapt_weights, print_current
      print *, "*** Test decay process"
      print *
      print *, "* Initialization"
      call tao_random_create (rng, 0)
      call process_store_init_process &
           (process, prc_lib, var_str ("zqq"), model, lhapdf_status, &
            use_beams=.false.)
      print *, "  Process ID = ", char (process%id)
      print *
      print *, "*** Beam/strfun setup (unpolarized)"
      print *
      call flavor_init (flv, (/ 23 /), model)
      call polarization_init_unpolarized (pol(1), flv(1))
      call beam_data_init_decay (beam_data, flv, pol)
      call process_setup_beams (process, beam_data, 0, 0)
      call process_connect_strfun (process)
      print *
      print *, "* Phase space setup"
      call process_setup_phase_space (process, rebuild_phs, &
           phs_par, mapping_defaults, filename_out=var_str("zee.phs"))
      print *
      print *, "*** Test integration"
      print *, "* Grids setup"
```

```
      grid_parameters%stratified = .false.
      call process_setup_grids (process, grid_parameters, calls=1000)
      print *
      print *, "* 1 iteration with minimal number of calls"
      call process_results_write_header (process)
      do i = 1, 1
         discard_integrals = i==1
         adapt_grids = .true.
         adapt_weights = .true.
         print_current = .true.
         call process_integrate (process, rng, grid_parameters, &
              1, 1, 1, 9, &
              discard_integrals, adapt_grids, adapt_weights, print_current, &
              time_estimate)
      end do
      call process_results_write_footer (process)
      call process_write_time_estimate (process)
      print *
      print *, "* Process written to 'fort.62'"
      call process_write (process, 62)
      print *
      print *, "*** Event generation"
      call process_setup_event_generation (process)
      print *
      print *, "* Generate weighted event"
      call process_generate_weighted_event (process, rng, weight)
      print *
      print *, "* Process written to 'fort.63'"
      call process_write (process, 63)
      print *, "weight =", weight
      print *
      print *, "* Generate unweighted event"
      call process_generate_unweighted_event (process, rng, weight)
      print *
      print *, "* Process written to 'fort.64'"
      call process_write (process, 64)
      print *, "excess weight =", weight
    end subroutine process_test2
```

Test scattering process: ee-¿nnh (colorless)

⟨*Processes: procedures*⟩+≡

```
    subroutine process_test3 (prc_lib, model)
      type(process_library_t), intent(inout) :: prc_lib
      type(model_t), intent(in), target :: model
      type(lhapdf_status_t) :: lhapdf_status
      type(process_t), pointer :: process
      type(phs_parameters_t) :: phs_par
      type(mapping_defaults_t) :: mapping_defaults
      type(flavor_t), dimension(2) :: flv
      type(polarization_t), dimension(2) :: pol
      type(beam_data_t) :: beam_data
      type(grid_parameters_t) :: grid_parameters
      real(default), dimension(:), allocatable :: x
      logical :: time_estimate = .true.
```

```
integer :: channel
logical :: ok
integer :: i
type(tao_random_state) :: rng
logical :: rebuild_phs = .true.
logical :: discard_integrals, adapt_grids, adapt_weights, print_current
print *, "*** Test scattering process"
print *
print *, "* Initialization"
call tao_random_create (rng, 0)
call process_store_init_process &
     (process, prc_lib, var_str ("nnh"), model, lhapdf_status)
print *, "  Process ID = ", char (process%id)
print *
print *, "* Beam/strfun setup"
print *
call flavor_init (flv, (/ 11, -11 /), model)
call polarization_init_unpolarized (pol(1), flv(1))
call polarization_init_unpolarized (pol(2), flv(2))
call beam_data_init_sqrts (beam_data, 500._default, flv, pol)
call process_setup_beams (process, beam_data, 0, 0)
call process_connect_strfun (process)
print *
print *, "* Phase space setup"
call process_setup_phase_space (process, rebuild_phs, &
     phs_par, mapping_defaults, filename_out=var_str("nnh.phs"))
print *
print *, "* Kinematics setup"
allocate (x (process_get_n_parameters (process)))
do i = 1, size (x)
   x(i) = (i - 0.5_default) * (1._default / size (x))
end do
channel = 1
call process_set_kinematics (process, x, channel, ok)
print *
print *, "* Process written to 'fort.70'"
call process_write (process, 70)
print *
print *, "*** Test process evaluation"
call process_evaluate (process)
print *
print *, "* Process written to 'fort.71'"
call process_write (process, 71)
print *
print *, "*** Test integration"
print *, "* Grids setup"
call process_setup_grids (process, grid_parameters, calls=10000)
print *
print *, "* 5 + 3 iterations"
call process_results_write_header (process)
do i = 1, 5
   discard_integrals = i==1
   adapt_grids = .true.
   adapt_weights = .true.
```

```
            print_current = .true.
            call process_integrate (process, rng, grid_parameters, &
                  1, 1, 1, 10000, &
                  discard_integrals, adapt_grids, adapt_weights, print_current, &
                  time_estimate)
         end do
         call process_results_write_current_average (process)
         call process_integrate (process, rng, grid_parameters, &
               2, 1, 3, 20000, .true., .false., .false., .true., .true.)
         call process_results_write_footer (process)
         call process_write_time_estimate (process)
         print *
         print *, "* Process written to 'fort.72'"
         call process_write (process, 72)
       end subroutine process_test3
```

Test scattering process: $gg \rightarrow qq$ with $q = u, d$.

⟨*Processes: procedures*⟩+≡

```
       subroutine process_test4 (prc_lib, model)
         type(process_library_t), intent(inout) :: prc_lib
         type(model_t), intent(in), target :: model
         type(lhapdf_status_t) :: lhapdf_status
         type(process_t), pointer :: process
         type(phs_parameters_t) :: phs_par
         type(mapping_defaults_t) :: mapping_defaults
         type(flavor_t), dimension(2) :: flv
         type(polarization_t), dimension(2) :: pol
         type(beam_data_t) :: beam_data
         type(lhapdf_data_t), dimension(2) :: data
         type(stream_t), target :: stream
         type(parse_tree_t) :: parse_tree
         type(grid_parameters_t) :: grid_parameters
         logical :: time_estimate = .true.
         integer :: i
         type(tao_random_state) :: rng
         logical :: rebuild_phs = .true.
         print *, "*** Test process setup"
         print *
         print *, "* Initialization"
         call tao_random_create (rng, 0)
         call process_store_init_process &
               (process, prc_lib, var_str ("qq"), model, lhapdf_status)
         print *, "  Process ID = ", char (process%id)
         print *
         print *, "* Beam/strfun setup"
         print *
!        call flavor_init (flv, (/ 21, 21 /), model)
         call flavor_init (flv, (/ PROTON, PROTON /), model)
         call polarization_init_unpolarized (pol(1), flv(1))
         call polarization_init_unpolarized (pol(2), flv(2))
         call beam_data_init_sqrts (beam_data, 14000._default, flv, pol)
!          call process_setup_beams (process, beam_data, 0, 0)
         call process_setup_beams (process, beam_data, 2, 0)
         call lhapdf_data_init (data(1), lhapdf_status, model, flv(1))
```

1089

```
      call lhapdf_data_init (data(2), lhapdf_status, model, flv(2))
      call process_set_strfun (process, 1, 1, data(1), 1)
      call process_set_strfun (process, 2, 2, data(2), 1)
      call process_connect_strfun (process)
      print *
      print *, "* Phase space setup"
      call process_setup_phase_space (process, rebuild_phs, &
           phs_par, mapping_defaults, filename_out=var_str("qq.phs"))
      print *
      print *, "* Cuts setup"
      call stream_init (stream, var_str ("all Pt > 50 GeV (outgoing u:d:U:D)"))
      call parse_tree_init_lexpr (parse_tree, stream, .true.)
      call process_setup_cuts (process, parse_tree_get_root_ptr (parse_tree))
      call parse_tree_final (parse_tree)
      call stream_final (stream)
      print *
      print *, "* Scale setup"
      call stream_init (stream, var_str ("1 TeV"))
      call parse_tree_init_expr (parse_tree, stream, .true.)
      call process_setup_scale (process, parse_tree_get_root_ptr (parse_tree))
      call parse_tree_final (parse_tree)
      call stream_final (stream)
      print *
      print *, "*** Test integration"
      print *, "* Grids setup"
      call process_setup_grids (process, grid_parameters, calls=10000)
      print *
      print *, "* 15 + 3 iterations"
      call process_results_write_header (process)
      do i = 1, 15
         call process_integrate (process, rng, grid_parameters, &
              1, 1, 1, 50000, i==1, .true., i>2, .true., .true.)
      end do
      call process_results_write_current_average (process)
      call process_integrate (process, rng, grid_parameters, &
           2, 1, 3, 50000, .true., .false., .true., .true., .true.)
      call process_results_write_footer (process)
      call process_write_time_estimate (process)
      print *
      print *, "* Process written to 'fort.90'"
      call process_write (process, 90)
   end subroutine process_test4
```

13.4   Decays

Particles can be marked as unstable, so during event generation (cascade) decays are applied to them. For each decay mode, we temporarily use the corresponding process entry in the process store, which is filled and evaluated and connected to the mother process.

⟨decays.f90⟩≡
  ⟨*File header*⟩

```
module decays
```

⟨*Use kinds*⟩
```
  use kinds, only: double !NODEP!
```
⟨*Use strings*⟩
```
  use limits, only: MAX_TRIES_FOR_DECAY_CHAIN
```
⟨*Use file utils*⟩
```
  use diagnostics !NODEP!
  use lorentz !NODEP!
  use tao_random_numbers !NODEP!
  use md5
  use models
  use flavors
  use quantum_numbers
  use processes
  use interactions
  use evaluators
```

⟨*Standard module head*⟩

⟨*Decays: public*⟩

⟨*Decays: types*⟩

⟨*Decays: variables*⟩

```
contains
```

⟨*Decays: procedures*⟩

```
end module decays
```

13.4.1 Decay configuration

We store the decay properties of a particular particle. First, we need an array of process pointers. For the final-state particles, we need to store the fact whether they are stable themselves and, if not, their decay properties. This is necessary because it determines the quantum numbers we need to keep for the decay products in event generation.

⟨*Decays: types*⟩≡
```
    type :: decay_channel_t
       private
       type(process_t), pointer :: process => null ()
       real(default) :: br = 0
       type(flavor_t), dimension(:), allocatable :: unstable_products
       logical, dimension(:), allocatable :: isotropic
       logical, dimension(:), allocatable :: diagonal
    end type decay_channel_t
```

Decay configurations are stored in a list:

⟨*Decays: public*⟩≡
```
    public :: decay_configuration_t
```

⟨*Decays: types*⟩+≡
```
    type :: decay_configuration_t
      private
      type(flavor_t) :: flv
      type(model_t), pointer :: model => null ()
      real(default) :: width = 0
      logical :: isotropic = .false.
      logical :: diagonal = .false.
      type(decay_channel_t), dimension(:), allocatable :: channel
      type(string_t), dimension(:), allocatable :: process_id
      type(decay_configuration_t), pointer :: next => null ()
    end type decay_configuration_t
```

Allocate the array for a known number of decay channels.

⟨*Decays: procedures*⟩≡
```
    subroutine decay_configuration_init &
        (conf, flv, model, width, n_channels, isotropic, diagonal)
      type(decay_configuration_t), intent(out) :: conf
      type(flavor_t), intent(in) :: flv
      type(model_t), intent(in), target :: model
      real(default), intent(in) :: width
      integer, intent(in) :: n_channels
      logical, intent(in) :: isotropic, diagonal
      conf%flv = flv
      conf%model => model
      conf%width = width
      conf%isotropic = isotropic
      conf%diagonal = diagonal
      allocate (conf%channel (n_channels))
      allocate (conf%process_id (n_channels))
    end subroutine decay_configuration_init
```

Set/retrieve the pointer to the next configuration in list:

⟨*Decays: procedures*⟩+≡
```
    function decay_configuration_get_next_ptr (conf) result (ptr)
      type(decay_configuration_t), pointer :: ptr
      type(decay_configuration_t), intent(in) :: conf
      ptr => conf%next
    end function decay_configuration_get_next_ptr

    subroutine decay_configuration_set_next_ptr (conf, ptr)
      type(decay_configuration_t), intent(inout) :: conf
      type(decay_configuration_t), pointer :: ptr
      conf%next => ptr
    end subroutine decay_configuration_set_next_ptr
```

Set a single decay channel:

⟨*Decays: public*⟩+≡
```
    public :: decay_configuration_set_channel
```

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_configuration_set_channel (conf, i, process, br)
      type(decay_configuration_t), intent(inout) :: conf
```

```
      integer, intent(in) :: i
      type(process_t), intent(in), target :: process
      real(default), intent(in) :: br
      integer :: n_unstable_products
      conf%channel(i)%process => process
      conf%channel(i)%br = br
      conf%process_id(i) = process_get_id (process)
      call process_get_unstable_products &
           (conf%channel(i)%process, conf%channel(i)%unstable_products)
      n_unstable_products = size (conf%channel(i)%unstable_products)
      if (allocated (conf%channel(i)%isotropic)) &
           deallocate (conf%channel(i)%isotropic)
      if (allocated (conf%channel(i)%diagonal)) &
           deallocate (conf%channel(i)%diagonal)
      allocate (conf%channel(i)%isotropic (n_unstable_products))
      allocate (conf%channel(i)%diagonal (n_unstable_products))
      if (n_unstable_products /= 0) then
         conf%channel(i)%isotropic = &
              flavor_decays_isotropically (conf%channel(i)%unstable_products)
         conf%channel(i)%diagonal = &
              flavor_decays_diagonal (conf%channel(i)%unstable_products)
      end if
    end subroutine decay_configuration_set_channel
```

Check an existing decay configuration whether the stability of the decay products has changed. If yes, re-initialize event generation for the corresponding channel.

Do the check only if the configuration corresponds to a particle that is currently known as unstable.

⟨*Decays: procedures*⟩+≡

```
    subroutine decay_configuration_recheck_final_state (conf, verbose)
      type(decay_configuration_t), intent(inout) :: conf
      logical, intent(in), optional :: verbose
      type(flavor_t), dimension(:), allocatable :: flv_unstable
      logical, dimension(:), allocatable :: isotropic, diagonal
      logical :: modified, verb
      integer :: u, i, n_unstable_products
      u = logfile_unit ()
      verb = .false.;  if (present (verbose))  verb = verbose
      if (flavor_is_stable (conf%flv))  return
      do i = 1, size (conf%channel)
         call process_get_unstable_products &
            (conf%channel(i)%process, flv_unstable)
         n_unstable_products = size (flv_unstable)
         allocate (isotropic (n_unstable_products))
         allocate (diagonal (n_unstable_products))
         isotropic = flavor_decays_isotropically (flv_unstable)
         diagonal = flavor_decays_diagonal (flv_unstable)
         if (n_unstable_products == size (conf%channel(i)%unstable_products)) &
              then
            modified = &
                 any (flv_unstable /= conf%channel(i)%unstable_products) &
                 .or. &
```

1093

```
                     any (isotropic .neqv. conf%channel(i)%isotropic) &
                     .or. &
                     any (diagonal .neqv. conf%channel(i)%diagonal)
           else
              modified = .true.
              deallocate (conf%channel(i)%unstable_products)
              deallocate (conf%channel(i)%isotropic)
              deallocate (conf%channel(i)%diagonal)
              allocate (conf%channel(i)%unstable_products (n_unstable_products))
              allocate (conf%channel(i)%isotropic (n_unstable_products))
              allocate (conf%channel(i)%diagonal (n_unstable_products))
           end if
           if (modified) then
              conf%channel(i)%unstable_products = flv_unstable
              conf%channel(i)%isotropic = isotropic
              conf%channel(i)%diagonal = diagonal
              call process_setup_event_generation (conf%channel(i)%process, &
                   qn_mask_in = new_quantum_numbers_mask (.false., .false., &
                      mask_h =  conf%isotropic, mask_hd = conf%diagonal))
              if (verb) then
                 call msg_message ("Further modified decay configuration:")
                 call decay_configuration_write (conf)
                 call decay_configuration_write (conf, u)
              end if
           end if
           deallocate (flv_unstable, isotropic, diagonal)
        end do
      end subroutine decay_configuration_recheck_final_state
```

Output. Note that either no channel or all channels have to be defined.

⟨*Decays: public*⟩+≡
```
    public :: decay_configuration_write
```

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_configuration_write (conf, unit)
      type(decay_configuration_t), intent(in) :: conf
      integer, intent(in), optional :: unit
      character(12) :: fmt
      integer :: n_channels, proc_id_len
      integer :: u, i, j
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)")  "Decay configuration of particle '" &
           // char (flavor_get_name (conf%flv)) // "' in model '" &
           // char (model_get_name (conf%model)) // "':"
      write (u, *) " Computed total width = ", conf%width, " GeV"
      if (conf%isotropic) then
         write (u, *) " Isotropic decays requested for simulation."
      end if
      if (conf%diagonal) then
         write (u, *) " Diagonal density matrix in decays " &
              // "requested for simulation."
      end if
      write (u, *) " Branching ratios:"
      n_channels = decay_configuration_get_n_channels (conf)
```

```
      if (n_channels /= 0) then
         proc_id_len = maxval (len (conf%process_id))
         do i = 1, n_channels
            write (u, "(F12.7,1x,A)", advance="no") 100 * conf%channel(i)%br, "%"
            write (fmt, "(2x,A,I0,A)") "(4x,A", proc_id_len + 1, ")"
            write (u, fmt, advance="no")  char (conf%process_id(i))
            if (allocated (conf%channel(i)%unstable_products)) then
               if (size (conf%channel(i)%unstable_products) /= 0) then
                  write (u, "(1x,A)", advance="no") "   -> unstable:"
                  do j = 1, size (conf%channel(i)%unstable_products)
                     write (u, "(1x,A)", advance="no")  char (flavor_get_name &
                          (conf%channel(i)%unstable_products(j)))
                     if (conf%channel(i)%isotropic(j)) then
                        write (u, "(A)", advance="no")  "[I]"
                     else if (conf%channel(i)%diagonal(j)) then
                        write (u, "(A)", advance="no")  "[D]"
                     end if
                  end do
               end if
            end if
            write (u, *)
         end do
      else
         write (u, *) " [undefined]"
      end if
    end subroutine decay_configuration_write
```

Return the number of decay channels:

⟨Decays: public⟩+≡
```
    public :: decay_configuration_get_n_channels
```

⟨Decays: procedures⟩+≡
```
    function decay_configuration_get_n_channels (conf) result (n)
      integer :: n
      type(decay_configuration_t), intent(in) :: conf
      if (allocated (conf%channel)) then
         n = size (conf%channel)
      else
         n = 0
      end if
    end function decay_configuration_get_n_channels
```

Select a decay channel, using the random-number generator. Return the process
pointer. Note that the sum of branching ratios must be unity. (As a fallback,
the last channel is selected if the sum of ratios is less than unity.)

⟨Decays: procedures⟩+≡
```
    function decay_configuration_select_channel (conf, rng) result (channel)
      integer :: channel
      type(decay_configuration_t), intent(in) :: conf
      type(tao_random_state), intent(inout) :: rng
      real(double) :: x
      real(default) :: x_sum
      call tao_random_number (rng, x)
```

```
      x_sum = 0
      do channel = 1, size (conf%channel)
         x_sum = x_sum + conf%channel(channel)%br
         if (x < x_sum)  return
      end do
      channel = size (conf%channel)
    end function decay_configuration_select_channel
```

Return a pointer to a specified decay process.

⟨*Decays: procedures*⟩+≡
```
    function decay_configuration_get_process_ptr (conf, channel) &
         result (process)
      type(process_t), pointer :: process
      type(decay_configuration_t), intent(in) :: conf
      integer, intent(in) :: channel
      process => conf%channel(channel)%process
    end function decay_configuration_get_process_ptr
```

13.4.2 List of decay configurations

Similar to the list of active processes (`process_store`), we maintain a list of unstable particles and their decay properties.

⟨*Decays: types*⟩+≡
```
    type :: decay_store_t
      private
      integer :: n = 0
      type(decay_configuration_t), pointer :: first => null ()
      type(decay_configuration_t), pointer :: last => null ()
    end type decay_store_t
```

⟨*Decays: variables*⟩≡
```
    type(decay_store_t), save :: store
```

Finalize.

⟨*Decays: public*⟩+≡
```
    public :: decay_store_final
```

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_store_final ()
      type(decay_configuration_t), pointer :: current
      store%last => null ()
      do while (associated (store%first))
         current => store%first
         store%first => current%next
         deallocate (current)
      end do
      store%n = 0
    end subroutine decay_store_final
```

Output.

⟨*Decays: public*⟩+≡
```
    public :: decay_store_write
```

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_store_write (unit)
      integer, intent(in), optional :: unit
      type(decay_configuration_t), pointer :: decay
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)")  "Decay configuration for unstable particles:"
      decay => store%first
      do while (associated (decay))
         if (.not. flavor_is_stable (decay%flv)) &
              call decay_configuration_write (decay, unit)
         decay => decay%next
      end do
    end subroutine decay_store_write
```

Retrieve the MD5 sum of the decay store.

⟨*Decays: public*⟩+≡
```
    public :: decay_store_get_md5sum
```

⟨*Decays: procedures*⟩+≡
```
    function decay_store_get_md5sum () result (md5sum_decays)
      character(32) :: md5sum_decays
      integer :: u
      u = free_unit ()
      open (u, status="scratch")
      call decay_store_write (u)
      rewind (u)
      md5sum_decays = md5sum (u)
    end function decay_store_get_md5sum
```

Append a new entry for an unstable particle. If a decay exists already, it is overwritten. Return a pointer to the new decay configuration.

⟨*Decays: public*⟩+≡
```
    public :: decay_store_append_decay
```

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_store_append_decay &
         (flv, model, width, n_channels, isotropic, diagonal, decay)
      type(flavor_t), intent(in) :: flv
      type(model_t), intent(in), target :: model
      real(default), intent(in) :: width
      integer, intent(in) :: n_channels
      logical, intent(in) :: isotropic, diagonal
      type(decay_configuration_t), pointer :: decay
      type(decay_configuration_t), pointer :: next_decay
      decay => store%first
      do while (associated (decay))
         if (decay%flv == flv) then
            next_decay => decay_configuration_get_next_ptr (decay)
            call decay_configuration_init &
```

```
             (decay, flv, model, width, n_channels, isotropic, diagonal)
         call decay_configuration_set_next_ptr (decay, next_decay)
           return
       end if
       decay => decay%next
    end do
    allocate (decay)
    call decay_configuration_init &
         (decay, flv, model, width, n_channels, isotropic, diagonal)
    if (associated (store%first)) then
       store%last%next => decay
    else
       store%first => decay
    end if
    store%last => decay
  end subroutine decay_store_append_decay
```

Return a pointer to the decay configuration for a particular unstable particle.

⟨*Decays: procedures*⟩+≡
```
    function decay_store_get_decay_configuration_ptr (flv) result (config)
      type(decay_configuration_t), pointer :: config
      type(flavor_t), intent(in) :: flv
      config => store%first
      SCAN_PARTICLES: do while (associated (config))
          if (config%flv == flv)  exit SCAN_PARTICLES
          config => config%next
      end do SCAN_PARTICLES
    end function decay_store_get_decay_configuration_ptr
```

Recheck all decay configurations whether the stability of the final state has changed. If yes, re-initialize event generation for the corresponding decay process.

This should be executed anytime the stablity of a particle has changed.

⟨*Decays: public*⟩+≡
```
    public :: decay_store_recheck_final_state
```

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_store_recheck_final_state (verbose)
      logical, intent(in), optional :: verbose
      logical :: modified
      type(decay_configuration_t), pointer :: config
      config => store%first
      do while (associated (config))
          call decay_configuration_recheck_final_state (config, verbose)
          config => config%next
      end do
    end subroutine decay_store_recheck_final_state
```

13.4.3 Decays

The decay object contains a pointer to the decay process, evaluators that hold the product of production and decay, and a pointer to the next decay node

(which holds all possible subsequent decays).

⟨*Decays: types*⟩+≡

```
type :: decay_t
   private
   logical :: initialized = .false.
   type(process_t), pointer :: process => null ()
   type(evaluator_t) :: eval_sqme
   type(evaluator_t) :: eval_flows
   type(decay_node_t), pointer :: next_node => null ()
end type decay_t
```

Initialize the decay with a certain process, and use this together with the production evaluators to initialize product evaluators.

⟨*Decays: procedures*⟩+≡

```
subroutine decay_init (decay, process, eval_sqme, eval_flows, i)
  type(decay_t), intent(out), target :: decay
  type(process_t), intent(inout), target :: process
  type(evaluator_t), intent(in), target :: eval_sqme, eval_flows
  integer, intent(in) :: i
  type(interaction_t), pointer :: prc_int
  type(evaluator_t), pointer :: prc_eval_sqme, prc_eval_flows
  integer :: n_tot
  logical, dimension(:), allocatable :: ignore_hel
  type(quantum_numbers_mask_t), dimension(:), allocatable :: &
       mask_hel, mask_sqme, mask_flows
  type(quantum_numbers_mask_t) :: mask_conn
  call process_request_copy (process, decay%process)
  prc_int => process_get_hi_int_ptr (decay%process)
  prc_eval_sqme => process_get_hi_eval_sqme_ptr (decay%process)
  prc_eval_flows => process_get_hi_eval_flows_ptr (decay%process)
  n_tot = evaluator_get_n_tot (prc_eval_sqme)
  allocate (ignore_hel (n_tot))
  ignore_hel(1) = .true.
  ignore_hel(2:) = .false.
  allocate (mask_hel (n_tot), mask_sqme (n_tot), mask_flows (n_tot))
  call quantum_numbers_mask_set_helicity (mask_hel, ignore_hel)
  mask_sqme = evaluator_get_mask (prc_eval_sqme) .or. mask_hel
  mask_flows = evaluator_get_mask (prc_eval_flows) .or. mask_hel
  mask_conn = new_quantum_numbers_mask (.false., .false., .true.)
  call evaluator_set_source_link (prc_eval_sqme, 1, eval_sqme, i)
  call evaluator_set_source_link (prc_eval_flows, 1, eval_flows, i)
  call evaluator_init_product (decay%eval_sqme, &
       eval_sqme, prc_eval_sqme, mask_conn, &
       connections_are_resonant=.true.)
  call evaluator_init_product (decay%eval_flows, &
       eval_flows, prc_eval_flows, mask_conn, &
       connections_are_resonant=.true.)
  call evaluator_set_source_link (prc_eval_sqme, 1, prc_int, 1)
  call evaluator_set_source_link (prc_eval_flows, 1, prc_int, 1)
  allocate (decay%next_node)
  decay%initialized = .true.
end subroutine decay_init
```

Finalizer: Delete the evaluators. Do not delete the process copy yet.

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_final (decay)
  type(decay_t), intent(inout) :: decay
  if (decay%initialized) then
    if (associated (decay%next_node)) &
        call decay_node_final (decay%next_node)
    call evaluator_final (decay%eval_sqme)
    call evaluator_final (decay%eval_flows)
  end if
end subroutine decay_final
```

Output.

⟨*Decays: procedures*⟩+≡

```
subroutine decay_write (decay, unit)
  type(decay_t), intent(in) :: decay
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(A)")  repeat ("=", 72)
  write (u, "(A)")  "Decay process:"
  call process_write (decay%process, unit)
  write (u, "(A)")  repeat ("=", 72)
  write (u, "(A)")  "Combined sqme including color factors " &
      // "(process + decay):"
  call evaluator_write (decay%eval_sqme, unit)
  write (u, "(A)")  repeat ("-", 72)
  write (u, "(A)")  "Combined color flow coefficients " &
      // "(process + decay):"
  call evaluator_write (decay%eval_flows, unit)
end subroutine decay_write
```

Given preconfigured evaluators, generate an event.

We generate an unweighted event for the decay process. This is isotropic in the decaying particle rest frame. To prepare spin correlation selection, we renormalize the squared matrix element by the trace. (Thus, the sum of the matrix element values is unity.)

TODO: Any excess weight is collected (to avoid VAMP warnings) but not recorded anywhere.

⟨*Decays: procedures*⟩+≡

```
subroutine decay_generate (decay, rng, flv, p)
  type(decay_t), intent(inout) :: decay
  type(tao_random_state), intent(inout) :: rng
  type(flavor_t), intent(in) :: flv
  type(vector4_t), intent(in) :: p
  real(default) :: excess
  type(evaluator_t), pointer :: process_eval_sqme
  call process_set_beam_momenta (decay%process, (/ p /))
  call process_generate_unweighted_event (decay%process, rng, excess=excess)
  process_eval_sqme => process_get_eval_sqme_ptr (decay%process)
  call evaluator_normalize_by_trace (process_eval_sqme)
  call evaluator_receive_momenta (decay%eval_sqme)
```

```
      call evaluator_receive_momenta (decay%eval_flows)
      call evaluator_evaluate (decay%eval_sqme)
      call evaluator_evaluate (decay%eval_flows)
    end subroutine decay_generate
```

13.4.4   Decay trees

A decay tree is created during event generation. Each node holds the possible
decays as branches, together with the decay configuration which is used to select
a branch for a particular event. Whenever a branch is selected for the first time,
it is initialized with the appropriate evaluators, which are then kept for later
use.

⟨*Decays: types*⟩+≡
```
    type :: decay_node_t
      private
      type(decay_configuration_t), pointer :: configuration => null ()
      integer :: current_channel = 0
      type(decay_t), dimension(:), allocatable :: decay
    end type decay_node_t
```

Initializer:

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_node_init (node, flv)
      type(decay_node_t), intent(out) :: node
      type(flavor_t), intent(in) :: flv
      node%configuration => decay_store_get_decay_configuration_ptr (flv)
      if (associated (node%configuration)) then
         allocate (node%decay &
               (decay_configuration_get_n_channels (node%configuration)))
      else
         call msg_bug ("Particle '" // char (flavor_get_name (flv)) &
               // "': Missing decay configuration")
      end if
    end subroutine decay_node_init
```

Recursive finalizer:

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_node_final (node)
      type(decay_node_t), intent(inout) :: node
      integer :: i
      if (allocated (node%decay)) then
         do i = 1, size (node%decay)
            call decay_final (node%decay(i))
         end do
         deallocate (node%decay)
      end if
    end subroutine decay_node_final
```

Write the currently selected decay:

⟨*Decays: procedures*⟩+≡

```
      subroutine decay_node_write (node, unit)
        type(decay_node_t), intent(in) :: node
        integer, intent(in), optional :: unit
        integer :: channel, u
        u = output_unit (unit)
        write (u, "(A)")  "|" // repeat ("=", 79)
        if (associated (node%configuration)) then
           call decay_configuration_write (node%configuration, unit)
           channel = node%current_channel
           if (channel /= 0) then
              write (u, "(1x,A)", advance="no")  "Decay node: "
              write (u, *) "current channel = ", channel
              call decay_write (node%decay(channel), unit)
           else
              write (u, *)  "Decay node: [no channel selected]"
           end if
        else
           write (u, *)  "Decay configuration: [undefined]"
        end if
      end subroutine decay_node_write
```

Return a pointer to the currently selected decay, or null if absent:

⟨*Decays: procedures*⟩+≡

```
      function decay_node_get_next_ptr (node) result (ptr)
        type(decay_node_t), pointer :: ptr
        type(decay_node_t), intent(in) :: node
        if (node%current_channel /= 0) then
           ptr => node%decay(node%current_channel)%next_node
        else
           ptr => null ()
        end if
      end function decay_node_get_next_ptr
```

The decay tree holds references to the production process as well as pointers to the final evaluators.

⟨*Decays: public*⟩+≡

```
      public :: decay_tree_t
```

⟨*Decays: types*⟩+≡

```
      type :: decay_tree_t
         private
         integer :: tries = 0
         real(default) :: acceptance_probability = 0
         type(process_t), pointer :: hard_process => null ()
         type(evaluator_t), pointer :: eval_sqme_in => null ()
         type(evaluator_t), pointer :: eval_flows_in => null ()
         type(decay_node_t), pointer :: root => null ()
         type(evaluator_t), pointer :: eval_sqme => null ()
         type(evaluator_t), pointer :: eval_flows => null ()
      end type decay_tree_t
```

Initialize the decay tree with a particular process and allocate the root node.

⟨*Decays: public*⟩+≡

```
      public :: decay_tree_init
```

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_tree_init (decay_tree, process)
      type(decay_tree_t), intent(out) :: decay_tree
      type(process_t), intent(in), target :: process
      decay_tree%hard_process => process
      decay_tree%eval_sqme_in => process_get_eval_sqme_ptr (process)
      decay_tree%eval_flows_in => process_get_eval_flows_ptr (process)
      allocate (decay_tree%root)
    end subroutine decay_tree_init
```

⟨*Decays: public*⟩+≡
```
    public :: decay_tree_final
```

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_tree_final (decay_tree)
      type(decay_tree_t), intent(inout) :: decay_tree
      if (associated (decay_tree%root)) then
         call decay_node_final (decay_tree%root)
         deallocate (decay_tree%root)
      end if
    end subroutine decay_tree_final
```

Output.

⟨*Decays: public*⟩+≡
```
    public :: decay_tree_write
```

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_tree_write (decay_tree, unit)
      type(decay_tree_t), intent(in) :: decay_tree
      integer, intent(in), optional :: unit
      type(decay_node_t), pointer :: decay_node
      integer :: u
      u = output_unit (unit)
      write (u, "(A)") "|" // repeat ("=", 79)
      write (u, *) "Decay tree:"
      write (u, *) "  tries = ", decay_tree%tries
      write (u, *) "  acceptance probability = ", &
             decay_tree%acceptance_probability
      write (u, "(A)") "|" // repeat ("=", 79)
      write (u, "(1x,A)", advance="no") "Mother process = "
      if (associated (decay_tree%hard_process)) then
         write (u, "(A)")  "'" &
             // char (process_get_id (decay_tree%hard_process)) &
             // "'"
      else
         write (u, "(A)")  "[undefined]"
      end if
      write (u, "(A)") "|" // repeat ("=", 79)
      decay_node => decay_tree%root
      if (associated (decay_node)) then
         write (u, *) "Decay chain:"
         do while (associated (decay_node))
            call decay_node_write (decay_node, unit)
```

```
            decay_node => decay_node_get_next_ptr (decay_node)
         end do
      else
         write (u, *) "[No decays]"
      end if
      write (u, "(A)") "|" // repeat ("=", 79)
      write (u, "(1x,A)") "Evaluator: " &
           // "Color-summed including all decays"
      if (associated (decay_tree%eval_sqme)) then
         call evaluator_write (decay_tree%eval_sqme, unit)
      else
         write (u, "(A)")  "[undefined]"
      end if
      write (u, "(A)") "|" // repeat ("=", 79)
      write (u, "(1x,A)") "Evaluator: " &
           // "Color flow components including all decays"
      if (associated (decay_tree%eval_flows)) then
         call evaluator_write (decay_tree%eval_flows, unit)
      else
         write (u, "(A)")  "[undefined]"
      end if
      write (u, "(A)") "|" // repeat ("=", 79)
   end subroutine decay_tree_write
```

Generate a decay chain; construct the decay tree as far as necessary, otherwise reuse it.

To prepare for spin correlation selection, we renormalize the matrix elements of the parent interaction by the entry with maximum value. Normalizing all decay process appropriately (by the trace of their respective matrix elements), the (trace of the) product of the evaluators should result in a single value between zero and one. This is used as the probability for accepting the decay chain. If the chain is rejected, a new one is generated.

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: MAX_TRIES_FOR_DECAY_CHAIN = 100000
```

⟨*Decays: public*⟩+≡
```
    public :: decay_tree_generate_event
```

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_tree_generate_event (decay_tree, rng)
      type(decay_tree_t), intent(inout) :: decay_tree
      type(tao_random_state), intent(inout) :: rng
      real(default) :: x_decay
      real(double) :: x
      integer :: i
      call evaluator_normalize_by_max (decay_tree%eval_sqme_in)
      REJECTION: do i = 1, MAX_TRIES_FOR_DECAY_CHAIN
         decay_tree%tries = i
         decay_tree%eval_sqme => decay_tree%eval_sqme_in
         decay_tree%eval_flows => decay_tree%eval_flows_in
         call decay_node_generate_event (decay_tree%root)
         x_decay = evaluator_sum (decay_tree%eval_sqme)
         decay_tree%acceptance_probability = x_decay
         call tao_random_number (rng, x)
```

```
        if (x <= x_decay)  return
      end do REJECTION
      write (msg_buffer, "(A,I0,A)") "Failed to generate a decay chain " &
           // "after ", MAX_TRIES_FOR_DECAY_CHAIN, " tries"
      call msg_fatal ()
    contains
      recursive subroutine decay_node_generate_event (node)
        type(decay_node_t), intent(inout), target :: node
        type(flavor_t) :: flv
        type(vector4_t) :: p
        integer :: i, channel
        type(process_t), pointer :: process
        call evaluator_get_unstable_particle (decay_tree%eval_sqme, flv, p, i)
        if (flavor_is_defined (flv)) then
           if (.not. associated (node%configuration)) &
                call decay_node_init (node, flv)
           channel = decay_configuration_select_channel (node%configuration, rng)
           node%current_channel = channel
           if (.not. node%decay(channel)%initialized) then
              process => decay_configuration_get_process_ptr &
                   (node%configuration, channel)
              call decay_init (node%decay(channel), &
                   process, decay_tree%eval_sqme, decay_tree%eval_flows, i)
           end if
           call decay_generate (node%decay(channel), rng, flv, p)
           decay_tree%eval_sqme => node%decay(channel)%eval_sqme
           decay_tree%eval_flows => node%decay(channel)%eval_flows
           call decay_node_generate_event (node%decay(channel)%next_node)
        end if
      end subroutine decay_node_generate_event
    end subroutine decay_tree_generate_event
```

Return pointers to the final evaluators:

⟨*Decays: public*⟩+≡
```
    public :: decay_tree_get_eval_sqme_ptr
    public :: decay_tree_get_eval_flows_ptr
```

⟨*Decays: procedures*⟩+≡
```
    function decay_tree_get_eval_sqme_ptr (decay_tree) result (eval)
      type(evaluator_t), pointer :: eval
      type(decay_tree_t), intent(in), target :: decay_tree
      eval => decay_tree%eval_sqme
    end function decay_tree_get_eval_sqme_ptr

    function decay_tree_get_eval_flows_ptr (decay_tree) result (eval)
      type(evaluator_t), pointer :: eval
      type(decay_tree_t), intent(in), target :: decay_tree
      eval => decay_tree%eval_flows
    end function decay_tree_get_eval_flows_ptr
```

13.5 Events

The event record becomes relevant only after cross sections have been integrated. It gets filled by some signal process (including beam/structure functions) if an event has been successfully been generated (passing rejection).

If requested, particles in the event are subject to decay and/or showering. This is not implemented yet.

⟨events.f90⟩≡
  ⟨*File header*⟩

  module events

  ⟨*Use kinds*⟩
    use kinds, only: double !NODEP!
  ⟨*Use strings*⟩
    use limits, only: RAW_EVENT_FILE_VERSION !NODEP!
  ⟨*Use file utils*⟩
    use diagnostics !NODEP!
    use tao_random_numbers !NODEP!
    use os_interface
    use lexers
    use parser
    use prt_lists
    use expressions
    use models
    use flavors
    use state_matrices
    use polarizations
    use event_formats
    use hepmc_interface
    use particles
    use interactions
    use evaluators
    use process_libraries
    use beams
    use sf_lhapdf
    use mappings
    use phs_forests
    use cascades
    use processes
    use decays

  ⟨*Standard module head*⟩

  ⟨*Events: public*⟩

  ⟨*Events: types*⟩

  contains

  ⟨*Events: procedures*⟩

  end module events

1106

13.5.1 The event type

⟨Events: public⟩≡
```
    public :: event_t
```

⟨Events: types⟩≡
```
    type :: event_t
       type(process_t), pointer :: process => null ()
       type(decay_tree_t), pointer :: decay_tree => null ()
       logical :: particle_set_exists = .false.
       type(particle_set_t) :: particle_set
       real(default), pointer :: weight => null ()
       real(default) :: excess = 0
    end type event_t
```

The event record is initialized with a pointer to a specific "signal" process. The particle set is not (yet) initialized, this is done for each event. The event weight is presented as a target, so the event record acquires a pointer to this target. The same target is available to the analysis evaluation tree.

⟨Events: public⟩+≡
```
    public :: event_init
```

⟨Events: procedures⟩≡
```
    subroutine event_init (event, process, event_weight, decay_tree)
      type(event_t), intent(out) :: event
      type(process_t), intent(in), target :: process
      real(default), intent(in), target :: event_weight
      type(decay_tree_t), intent(in), optional, target :: decay_tree
      event%process => process
      event%weight => event_weight
      if (present (decay_tree))  event%decay_tree => decay_tree
    end subroutine event_init
```

Finalize the event: delete the particle set.

⟨Events: public⟩+≡
```
    public :: event_final
```

⟨Events: procedures⟩+≡
```
    subroutine event_final (event)
      type(event_t), intent(inout) :: event
      call particle_set_final (event%particle_set)
    end subroutine event_final
```

Output: Only the particle set is printed explicitly, unless verbose format is selected.

⟨Events: public⟩+≡
```
    public :: event_write
```

⟨Events: procedures⟩+≡
```
    subroutine event_write (event, unit, verbose)
      type(event_t), intent(in) :: event
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer :: u
```

```
      u = output_unit (unit);  if (u < 0)  return
      write (u, *) repeat ("=", 72)
      write (u, *) "Event record:"
      if (associated (event%process)) then
         if (present (verbose)) then
            if (verbose) then
               call process_write (event%process, unit)
               write (u, *)
            end if
         end if
      else
         write (u, *) "  [empty]"
      end if
      write (u, *) repeat ("=", 72)
      if (associated (event%decay_tree)) then
         write (u, *) repeat ("=", 72)
         call decay_tree_write (event%decay_tree, unit)
      end if
      write (u, *) "  [Process: ", char (process_get_id (event%process)), "]"
      write (u, *)
      call particle_set_write (event%particle_set, unit)
      write (u, *)
      if (associated (event%weight)) then
         write (u, *) "Event weight  =", event%weight
      else
         write (u, *) "Event weight  = [undefined]"
      end if
      write (u, *) "Excess weight =", event%excess
      write (u, *) repeat ("=", 72)
    end subroutine event_write
```

13.5.2   Event generation

Generate a new event and transfer the resulting data to the event record.

⟨*Events: public*⟩+≡
```
    public :: event_generate
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_generate (event, rng, unweighted, &
        factorization_mode, keep_correlations, keep_virtual)
      type(event_t), intent(inout), target :: event
      type(tao_random_state), intent(inout) :: rng
      logical, intent(in) :: unweighted
      integer, intent(in) :: factorization_mode
      logical, intent(in) :: keep_correlations, keep_virtual
      real(double) :: r
      call event_discard_particle_set (event)
      if (unweighted) then
         call process_generate_unweighted_event (event%process, rng, event%excess)
         event%weight = 1
      else
         call process_generate_weighted_event (event%process, rng, event%weight)
      end if
```

```
      if (associated (event%decay_tree)) then
         call decay_tree_generate_event (event%decay_tree, rng)
         call event_factorize_process (event, rng, &
              factorization_mode, keep_correlations, keep_virtual)
      end if
   end subroutine event_generate
```

Transfer event data from the process record (if a decay has happened: the decay chain) to the event record, factorizing the correlated quantum-number state. We use both the colorless and the colored evaluators to determine the particle set. The factorization of the correlated state is done in one of three modes (unpolarized, definite helicity, generic one-particle density matrices); optionally, the fully correlated density matrix can also be transferred to the particle set.

⟨*Events: procedures*⟩+≡
```
   subroutine event_factorize_process (event, rng, &
        factorization_mode, keep_correlations, keep_virtual)
     type(event_t), intent(inout), target :: event
     type(tao_random_state), intent(inout) :: rng
     integer, intent(in) :: factorization_mode
     logical, intent(in) :: keep_correlations, keep_virtual
     type(interaction_t), pointer :: int_sqme, int_flows
     real(double), dimension(2) :: r
     integer, dimension(:), allocatable :: beam_index
     integer, dimension(:), allocatable :: incoming_parton_index
     int_sqme => evaluator_get_int_ptr &
         (decay_tree_get_eval_sqme_ptr  (event%decay_tree))
     int_flows => evaluator_get_int_ptr &
         (decay_tree_get_eval_flows_ptr (event%decay_tree))
     call tao_random_number (rng, r)
     if (interaction_get_n_in (int_sqme) /= 0) then
        call particle_set_init (event%particle_set, &
             int_sqme, int_flows, factorization_mode, r, &
             keep_correlations, keep_virtual)
     else
        call particle_set_init (event%particle_set, &
             int_sqme, int_flows, factorization_mode, r, &
             keep_correlations, keep_virtual, &
             n_incoming = process_get_n_in (event%process))
     end if
     call process_get_beam_index (event%process, beam_index)
     if (allocated (beam_index)) then
        call particle_set_reset_status (event%particle_set, &
             beam_index, PRT_BEAM)
     end if
     call process_get_incoming_parton_index (event%process, &
          incoming_parton_index)
     if (allocated (incoming_parton_index)) then
        call particle_set_reset_status (event%particle_set, &
             incoming_parton_index, PRT_INCOMING)
     end if
     event%particle_set_exists = .true.
   end subroutine event_factorize_process
```

Analyze an event. The `prt_list` object is used as a messenger object to store the event particle data; the analysis expression has to be initialized before with reference to this object. The analysis results are stored as side-effect operations.

⟨*Events: public*⟩+≡
```
public :: event_do_analysis
```

⟨*Events: procedures*⟩+≡
```
subroutine event_do_analysis (event, prt_list, analysis_expr)
  type(event_t), intent(inout), target :: event
  type(prt_list_t), intent(inout), target :: prt_list
  type(eval_tree_t), intent(inout), target :: analysis_expr
  if (eval_tree_is_defined (analysis_expr)) then
     call particle_set_to_prt_list (event%particle_set, prt_list)
     call eval_tree_evaluate (analysis_expr)
  end if
end subroutine event_do_analysis
```

Delete any previous contents of the particle set.

⟨*Events: procedures*⟩+≡
```
subroutine event_discard_particle_set (event)
  type(event_t), intent(inout), target :: event
  if (event%particle_set_exists) then
     call particle_set_final (event%particle_set)
     event%particle_set_exists = .false.
  end if
end subroutine event_discard_particle_set
```

13.5.3 Binary I/O

Read/write the particle set including the associated state matrix from/to an unformatted file. This can be used to re-read events generated in a previous run.

⟨*Limits: public parameters*⟩+≡
```
integer, parameter, public :: RAW_EVENT_FILE_VERSION = 1
```

⟨*Events: public*⟩+≡
```
public :: raw_event_file_write_header
public :: raw_event_file_read_header
```

⟨*Events: procedures*⟩+≡
```
subroutine raw_event_file_write_header (unit, &
     md5sum_process, md5sum_parameters, md5sum_results, &
     md5sum_decays, md5sum_simulation)
  integer, intent(in) :: unit
  character(32), dimension(:), intent(in) :: md5sum_process
  character(32), dimension(:), intent(in) :: md5sum_parameters
  character(32), dimension(:), intent(in) :: md5sum_results
  character(32), intent(in) :: md5sum_decays, md5sum_simulation
  write (unit)  RAW_EVENT_FILE_VERSION
  write (unit)  size (md5sum_process)
  write (unit)  md5sum_process
  write (unit)  md5sum_parameters
  write (unit)  md5sum_results
```

```
    write (unit)  md5sum_decays
    write (unit)  md5sum_simulation
  end subroutine raw_event_file_write_header

  subroutine raw_event_file_read_header (unit, &
       md5sum_process, md5sum_parameters, md5sum_results, &
       md5sum_decays, md5sum_simulation, ok, iostat)
    integer, intent(in) :: unit
    character(32), dimension(:), intent(in) :: md5sum_process
    character(32), dimension(:), intent(in) :: md5sum_parameters
    character(32), dimension(:), intent(in) :: md5sum_results
    character(32), intent(in) :: md5sum_decays, md5sum_simulation
    logical, intent(out) :: ok
    integer, intent(out), optional :: iostat
    integer :: version, n
    character(32), dimension(:), allocatable :: md5sum_array
    character(32) :: md5sum
    logical :: unweighted
    ok = .false.
    read (unit, iostat=iostat)  version
    if (version /= RAW_EVENT_FILE_VERSION) then
       call msg_message &
            ("Event-file format version has changed, discarding old event file")
       return
    end if
    read (unit, iostat=iostat)  n
    if (n /= size (md5sum_process)) then
       call msg_message &
            ("Process number has changed, discarding old event file")
       return
    end if
    allocate (md5sum_array (n))
    read (unit, iostat=iostat)  md5sum_array
    if (any (md5sum_process /= md5sum_array)) then
       call msg_message &
            ("Process configuration has changed, discarding old event file")
       return
    end if
    read (unit, iostat=iostat)  md5sum_array
    if (any (md5sum_parameters /= md5sum_array)) then
       call msg_message &
            ("Model parameters have changed, discarding old event file")
       return
    end if
    read (unit, iostat=iostat)  md5sum_array
    if (any (md5sum_results /= md5sum_array)) then
       call msg_message &
            ("Integration results have changed, skipping event file")
       return
    end if
    read (unit, iostat=iostat)  md5sum
    if (md5sum_decays /= md5sum) then
       call msg_message &
            ("Decay configuration has changed, skipping event file")
```

1111

```
            return
         end if
         read (unit, iostat=iostat)  md5sum
         if (md5sum_simulation /= md5sum) then
            call msg_message &
                 ("Simulation parameters have changed, skipping event file")
            return
         end if
         ok = .true.
      end subroutine raw_event_file_read_header
```

⟨*Events: public*⟩+≡
```
      public :: event_write_raw
      public :: event_read_raw
```

⟨*Events: procedures*⟩+≡
```
      subroutine event_write_raw (event, unit)
        type(event_t), intent(in) :: event
        integer, intent(in) :: unit
        if (associated (event%process)) then
           write (unit)  process_get_store_index (event%process)
           write (unit)  process_get_scale (event%process)
           write (unit)  process_get_alpha_s (event%process)
           write (unit)  process_get_sqme (event%process)
        else
           write (unit)  0
           write (unit)  0._default
           write (unit)  0._default
           write (unit)  0._default
        end if
        call particle_set_write_raw (event%particle_set, unit)
        write (unit)  event%weight, event%excess
      end subroutine event_write_raw

      subroutine event_read_raw (event, unit, event_weight, iostat)
        type(event_t), intent(out) :: event
        integer, intent(in) :: unit
        real(default), intent(in), target :: event_weight
        integer, intent(out), optional :: iostat
        integer :: index
        real(default) :: scale, alpha_s, sqme
        read (unit, iostat=iostat)  index
        if (iostat /= 0) return
        read (unit, iostat=iostat)  scale
        read (unit, iostat=iostat)  alpha_s
        read (unit, iostat=iostat)  sqme
        event%process => process_store_get_process_ptr (index)
        call particle_set_read_raw (event%particle_set, unit, iostat=iostat)
        event%particle_set_exists = .true.
        if (associated (event%process)) then
           call process_set_particles (event%process, event%particle_set)
           call process_set_scale (event%process, scale)
           call process_set_alpha_s (event%process, alpha_s)
           call process_set_sqme (event%process, sqme)
```

```
      end if
      event%weight => event_weight
      read (unit, iostat=iostat)  event%weight, event%excess
   end subroutine event_read_raw
```

13.5.4 HepMC interface

Read/write the particle set from/to a HepMC event record. This accesses only the particle set; process data are not modified.

The polarization mode must be known when reading from HepMC because the HepMC event record does not specify it.

⟨*Events: public*⟩+≡
```
    public :: event_read_from_hepmc
    public :: event_write_to_hepmc
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_read_from_hepmc (event, hepmc_event, polarization_mode)
      type(event_t), intent(inout) :: event
      type(hepmc_event_t), intent(in) :: hepmc_event
      integer, intent(in) :: polarization_mode
      call event_discard_particle_set (event)
      call particle_set_init (event%particle_set, hepmc_event, &
           process_get_model_ptr (event%process), polarization_mode)
      event%particle_set_exists = .true.
    end subroutine event_read_from_hepmc

    subroutine event_write_to_hepmc (event, hepmc_event)
      type(event_t), intent(in) :: event
      type(hepmc_event_t), intent(inout) :: hepmc_event
      call particle_set_fill_hepmc_event (event%particle_set, hepmc_event)
    end subroutine event_write_to_hepmc
```

13.5.5 LHEF and HEPEVT interface

Fill the HEPEUP (event) common block:

⟨*Events: public*⟩+≡
```
    public :: event_write_to_hepeup
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_write_to_hepeup (event)
      type(event_t), intent(in) :: event
      integer :: proc_id
      real(default) :: scale, alpha_qcd
      call particle_set_fill_hepeup (event%particle_set)
      if (associated (event%process)) then
         proc_id = process_get_lib_index (event%process)
         call hepeup_set_event_parameters (proc_id = proc_id)
         scale = process_get_scale (event%process)
         if (scale /= 0)  call hepeup_set_event_parameters (scale = scale)
         alpha_qcd = process_get_alpha_s (event%process)
         if (alpha_qcd /= 0) &
```

```
              call hepeup_set_event_parameters (alpha_qcd = alpha_qcd)
            call hepeup_set_event_parameters (weight = event%weight)
        end if
      end subroutine event_write_to_hepeup
```

Fill the HEPEVT (event) common block:

⟨*Events: public*⟩+≡
```
    public :: event_write_to_hepevt
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_write_to_hepevt (event, keep_beams, i_evt)
      type(event_t), intent(in) :: event
      type(particle_set_t), target :: pset_reduced
      integer :: proc_id, n_tot, n_out, n_remnants
      integer, intent(in), optional :: i_evt
      logical, intent(in), optional :: keep_beams
      logical :: kb
      integer :: evt_count
      real(default) :: weight, function_value
      kb = .false.
      evt_count = 0
      if (present (keep_beams)) kb = keep_beams
      if (present (i_evt)) evt_count = i_evt
      call particle_set_fill_hepevt (event%particle_set, kb)
      call particle_set_reduce (event%particle_set, pset_reduced, kb)
      n_tot = particle_set_get_n_tot (pset_reduced)
      n_out = particle_set_get_n_out (pset_reduced)
      n_remnants = 0
      function_value = process_get_sample_function_value (event%process)
      call hepevt_set_event_parameters (n_tot, n_out, &
          n_remnants, weight = event%weight, &
          function_value = function_value, i_evt = evt_count)
    end subroutine event_write_to_hepevt
```

13.5.6   Test

⟨*Events: public*⟩+≡
```
    public :: event_test
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_test ()
      type(os_data_t) :: os_data
      type(process_library_t) :: prc_lib
      type(event_t), target :: event
      type(model_t), pointer :: model
      print *, "*** Read model file"
      call syntax_model_file_init ()
      call model_list_read_model &
          (var_str("QCD"), var_str("test.mdl"), os_data, model)
      call syntax_pexpr_init ()
      call syntax_phs_forest_init ()
      print *
      print *, "*** Load process library"
```

```
      call process_library_init (prc_lib, var_str("proc"), os_data)
      call process_library_load (prc_lib, os_data)
      print *
      call event_test1 (prc_lib, model)
      print *
      print *, "* Cleanup"
      call event_final (event)
      call process_store_final ()
      call syntax_pexpr_final ()
      call syntax_phs_forest_final ()
      call syntax_model_file_final ()
    end subroutine event_test
```

⟨*Events: procedures*⟩+≡

```
    subroutine event_test1 (prc_lib, model)
      type(process_library_t), intent(inout) :: prc_lib
      type(model_t), intent(in), target :: model
      type(lhapdf_status_t) :: lhapdf_status
      type(process_t), pointer :: process
      type(phs_parameters_t) :: phs_par
      type(mapping_defaults_t) :: mapping_defaults
      type(flavor_t), dimension(2) :: flv
      type(polarization_t), dimension(2) :: pol
      type(beam_data_t) :: beam_data
      type(stream_t), target :: stream
      type(parse_tree_t) :: parse_tree
      type(grid_parameters_t) :: grid_parameters
      integer :: i
      type(tao_random_state) :: rng
      type(event_t), target :: event
      real(default), target :: event_weight = 0
      type(decay_tree_t), target :: decay_tree
      logical :: rebuild_phs = .true.
      print *, "*** Test process setup"
      print *
      print *, "* Initialization"
      call tao_random_create (rng, 0)
      call process_store_init_process &
           (process, prc_lib, var_str ("qq"), model, lhapdf_status)
      print *, "  Process ID = ", char (process_get_id (process))
      print *
      print *, "* Beam setup"
      print *
      call flavor_init (flv, (/ 21, 21 /), model)
      call polarization_init_unpolarized (pol(1), flv(1))
      call polarization_init_unpolarized (pol(2), flv(2))
      call beam_data_init_sqrts (beam_data, 1000._default, flv, pol)
      call process_setup_beams (process, beam_data, 0, 0)
      call process_connect_strfun (process)
      print *
      print *, "* Phase space setup"
      call process_setup_phase_space (process, rebuild_phs, &
           phs_par, mapping_defaults, filename_out=var_str("qq.phs"))
      print *
```

```
      print *, "* Cuts setup"
      call stream_init (stream, var_str ("all Pt > 200 GeV (outgoing u:d:U:D)"))
      call parse_tree_init_lexpr (parse_tree, stream, .true.)
      call process_setup_cuts (process, parse_tree_get_root_ptr (parse_tree))
      call parse_tree_final (parse_tree)
      call stream_final (stream)
      print *
      print *, "*** Integration"
      print *, "* Grids setup"
      call process_setup_grids (process, grid_parameters, calls=10000)
      print *
      print *, "* 5 + 3 iterations"
      call process_results_write_header (process)
      do i = 1, 5
         call process_integrate (process, rng, grid_parameters, &
              1, 1, 1, 5000, i==1, .true., i>2, .true., .true.)
      end do
      call process_results_write_current_average (process)
      call process_integrate (process, rng, grid_parameters, &
           2, 1, 3, 5000, .true., .false., .true., .true., .true.)
      call process_results_write_footer (process)
      call process_write_time_estimate (process)
      print *
      print *, "*** Event generation"
      call process_setup_event_generation (process)
      call decay_tree_init (decay_tree, process)
      call event_init (event, process, event_weight, decay_tree)
      print *
      print *, "* Weighted event"
      call event_generate &
           (event, rng, .false., FM_IGNORE_HELICITY, .false., .false.)
      call event_write (event)
      print *
      print *, "* Unweighted event"
      call event_generate &
           (event, rng, .true., FM_SELECT_HELICITY, .false., .true.)
      call event_write (event)
      print *, "  Process data written to fort.81"
      call process_write (process, 81)
      call event_final (event)
    end subroutine event_test1
```

Chapter 14

External modules

14.1 STDHEP interface

Interface to the **STDHEP** library for portable I/O of binary event data. The
dummy module is needed if the STDHEP library cannot be linked, in order to
provide dummy routines. The second module contains calls to the actual stdhep
routines. The number of expected events is not really important. It is a 32-bit
number, so if the actual number gets larger, insert the maximal possible 32-bit
number.

When writing events, the flag `ilbl` determines the common block to write:
1 for HEPEVT, 11 for HEPEUP, 12 for HEPRUP

⟨**stdhep_interface.f90**⟩≡
  ⟨*File header*⟩

```
  module stdhep_interface

    use kinds, only: i32, i64 !NODEP!
```

⟨*Standard module head*⟩

```
    public :: stdhep_init, stdhep_write, stdhep_end

    integer, parameter, public :: &
         STDHEP_HEPEVT = 1, STDHEP_HEPEUP = 11, STDHEP_HEPRUP = 12

    integer, save :: istr, lok

  contains

    subroutine stdhep_init (file, title, nevt)
      character(len=*), intent(in) :: file, title
      integer(i64), intent(in) :: nevt
      integer(i32) :: nevt32
      external stdxwinit, stdxwrt
      nevt32 = min (nevt, int (huge (1_i32), i64))
      call stdxwinit (file, title, nevt32, istr, lok)
      call stdxwrt (100, istr, lok)
    end subroutine stdhep_init
```

```fortran
  subroutine stdhep_write (ilbl)
    integer, intent(in) :: ilbl
    external stdxwrt
    call stdxwrt (ilbl, istr, lok)
  end subroutine stdhep_write

  subroutine stdhep_end
    external stdxend
    call stdxend (istr)
  end subroutine stdhep_end

end module stdhep_interface
```

Chapter 15

The SUSY Les Houches Accord

The SUSY Les Houches Accord defines a standard interfaces for storing the physics data of SUSY models. Here, we provide the means for reading, storing, and writing such data.

⟨`slha_interface.f90`⟩≡
  ⟨*File header*⟩

  ```
  module slha_interface
  ```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ```
    use limits, only: EOF, VERSION_STRING !NODEP!
    use constants !NODEP!
  ```
  ⟨*Use file utils*⟩
  ```
    use diagnostics !NODEP!
    use os_interface
    use ifiles
    use lexers
    use syntax_rules
    use parser
    use variables
    use expressions
    use models
  ```

  ⟨*Standard module head*⟩

  ⟨*SLHA: public*⟩

  ⟨*SLHA: parameters*⟩

  ⟨*SLHA: variables*⟩

  ```
    save
  ```

  ```
  contains
  ```

1119

⟨*SLHA: procedures*⟩

```
end module slha_interface
```

15.0.1 Preprocessor

SLHA is a mixed-format standard. It should be read in assuming free format (but line-oriented), but it has some fixed-format elements.

To overcome this difficulty, we implement a preprocessing step which transforms the SLHA into a format that can be swallowed by our generic free-format lexer and parser. Each line with a blank first character is assumed to be a data line. We prepend a 'DATA' keyword to these lines. Furthermore, to enforce line-orientation, each line is appended a '$' key which is recognized by the parser. To do this properly, we first remove trailing comments, and skip lines consisting only of comments.

The preprocessor reads from a stream and puts out an `ifile`. Blocks that are not recognized are skipped. For some blocks, data items are quoted, so they can be read as strings if necessary.

⟨*SLHA: parameters*⟩≡
```
    integer, parameter :: MODE_SKIP = 0, MODE_DATA = 1, MODE_INFO = 2
```

⟨*SLHA: procedures*⟩≡
```
    subroutine slha_preprocess (stream, ifile)
      type(stream_t), intent(inout), target :: stream
      type(ifile_t), intent(out) :: ifile
      type(string_t) :: buffer, line, item
      integer :: iostat
      integer :: mode
      mode = MODE
      SCAN_FILE: do
         call stream_get_record (stream, buffer, iostat)
         select case (iostat)
         case (0)
            call split (buffer, line, "#")
            if (len_trim (line) == 0)  cycle SCAN_FILE
            select case (char (extract (line, 1, 1)))
            case ("B", "b")
               mode = check_block_handling (line)
               call ifile_append (ifile, line // "$")
            case ("D", "d")
               mode = MODE_DATA
               call ifile_append (ifile, line // "$")
            case (" ")
               select case (mode)
               case (MODE_DATA)
                  call ifile_append (ifile, "DATA" // line // "$")
               case (MODE_INFO)
                  line = adjustl (line)
                  call split (line, item, " ")
                  call ifile_append (ifile, "INFO" // " " // item // " " &
                       // '"' // trim (adjustl (line)) // '" $')
               end select
```

```
              case default
                 call msg_message (char (line))
                 call msg_fatal ("SLHA: Incomprehensible line")
              end select
           case (EOF)
              exit SCAN_FILE
           case default
              call msg_fatal ("SLHA: I/O error occured while reading SLHA input")
           end select
        end do SCAN_FILE
     end subroutine slha_preprocess
```

Return the mode that we should treat this block with. We need to recognize
only those blocks that we actually use.

⟨*SLHA: procedures*⟩+≡

```
    function check_block_handling (line) result (mode)
      integer :: mode
      type(string_t), intent(in) :: line
      type(string_t) :: buffer, key, block_name
      buffer = trim (line)
      call split (buffer, key, " ")
      buffer = adjustl (buffer)
      call split (buffer, block_name, " ")
      block_name = trim (adjustl (upper_case (block_name)))
      select case (char (block_name))
      case ("MODSEL", "MINPAR", "SMINPUTS")
         mode = MODE_DATA
      case ("MASS")
         mode = MODE_DATA
      case ("NMIX", "UMIX", "VMIX", "STOPMIX", "SBOTMIX", "STAUMIX")
         mode = MODE_DATA
      case ("NMHMIX", "NMAMIX", "NMNMIX", "NMSSMRUN")
         mode = MODE_DATA
      case ("ALPHA", "HMIX")
         mode = MODE_DATA
      case ("AU", "AD", "AE")
         mode = MODE_DATA
      case ("SPINFO", "DCINFO")
         mode = MODE_INFO
      case default
         mode = MODE_SKIP
      end select
    end function check_block_handling
```

15.0.2 Lexer and syntax

⟨*SLHA: variables*⟩≡

```
    type(syntax_t), target :: syntax_slha
```

⟨*SLHA: public*⟩≡

```
    public :: syntax_slha_init
```

⟨*SLHA: procedures*⟩+≡
```
    subroutine syntax_slha_init ()
      type(ifile_t) :: ifile
      call define_slha_syntax (ifile)
      call syntax_init (syntax_slha, ifile)
      call ifile_final (ifile)
    end subroutine syntax_slha_init
```

⟨*SLHA: public*⟩+≡
```
    public :: syntax_slha_final
```
⟨*SLHA: procedures*⟩+≡
```
    subroutine syntax_slha_final ()
      call syntax_final (syntax_slha)
    end subroutine syntax_slha_final
```

⟨*SLHA: public*⟩+≡
```
    public :: syntax_slha_write
```
⟨*SLHA: procedures*⟩+≡
```
    subroutine syntax_slha_write (unit)
      integer, intent(in), optional :: unit
      call syntax_write (syntax_slha, unit)
    end subroutine syntax_slha_write
```

⟨*SLHA: procedures*⟩+≡
```
    subroutine define_slha_syntax (ifile)
      type(ifile_t), intent(inout) :: ifile
      call ifile_append (ifile, "SEQ slha = chunk*")
      call ifile_append (ifile, "ALT chunk = block_def | decay_def")
      call ifile_append (ifile, "SEQ block_def = " &
           // "BLOCK block_spec '$' block_line*")
      call ifile_append (ifile, "KEY BLOCK")
      call ifile_append (ifile, "SEQ block_spec = block_name qvalue?")
      call ifile_append (ifile, "IDE block_name")
      call ifile_append (ifile, "SEQ qvalue = qname '=' real")
      call ifile_append (ifile, "IDE qname")
      call ifile_append (ifile, "KEY '='")
      call ifile_append (ifile, "REA real")
      call ifile_append (ifile, "KEY '$'")
      call ifile_append (ifile, "ALT block_line = block_data | block_info")
      call ifile_append (ifile, "SEQ block_data = DATA data_line '$'")
      call ifile_append (ifile, "KEY DATA")
      call ifile_append (ifile, "SEQ data_line = data_item+")
      call ifile_append (ifile, "ALT data_item = signed_number | number")
      call ifile_append (ifile, "SEQ signed_number = sign number")
      call ifile_append (ifile, "ALT sign = '+' | '-'")
      call ifile_append (ifile, "ALT number = integer | real")
      call ifile_append (ifile, "INT integer")
      call ifile_append (ifile, "KEY '-'")
      call ifile_append (ifile, "KEY '+'")
      call ifile_append (ifile, "SEQ block_info = INFO info_line '$'")
      call ifile_append (ifile, "KEY INFO")
      call ifile_append (ifile, "SEQ info_line = integer string_literal")
```

```
        call ifile_append (ifile, "QUO string_literal = '""'...'""'")
        call ifile_append (ifile, "SEQ decay_def = " &
             // "DECAY decay_spec '$' decay_data*")
        call ifile_append (ifile, "KEY DECAY")
        call ifile_append (ifile, "SEQ decay_spec = pdg_code data_item")
        call ifile_append (ifile, "ALT pdg_code = signed_integer | integer")
        call ifile_append (ifile, "SEQ signed_integer = sign integer")
        call ifile_append (ifile, "SEQ decay_data = DATA decay_line '$'")
        call ifile_append (ifile, "SEQ decay_line = data_item integer pdg_code+")
    end subroutine define_slha_syntax
```

The SLHA specification allows for string data items in certain places. Currently, we do not interpret them, but the strings, which are not quoted, must be parsed somehow. The hack for this problem is to allow essentially all characters as special characters, so the string can be read before it is discarded.

⟨*SLHA: public*⟩+≡
```
    public :: lexer_init_slha
```

⟨*SLHA: procedures*⟩+≡
```
    subroutine lexer_init_slha (lexer)
      type(lexer_t), intent(out) :: lexer
      call lexer_init (lexer, &
           comment_chars = "#", &
           quote_chars = '"', &
           quote_match = '"', &
           single_chars = "+-=$", &
           special_class = (/ "" /), &
           keyword_list = syntax_get_keyword_list_ptr (syntax_slha), &
           upper_case_keywords = .true.)  ! $
    end subroutine lexer_init_slha
```

15.0.3   Interpreter

Find blocks

From the parse tree, find the node that represents a particular block. If `required` is true, issue an error if not found. Since `block_name` is always invoked with capital letters, we have to capitalize `pn_block_name`.

⟨*SLHA: procedures*⟩+≡
```
    function slha_get_block_ptr &
         (parse_tree, block_name, required) result (pn_block)
      type(parse_node_t), pointer :: pn_block
      type(parse_tree_t), intent(in) :: parse_tree
      type(string_t), intent(in) :: block_name
      logical, intent(in) :: required
      type(parse_node_t), pointer :: pn_root, pn_block_spec, pn_block_name
      pn_root => parse_tree_get_root_ptr (parse_tree)
      pn_block => parse_node_get_sub_ptr (pn_root)
      do while (associated (pn_block))
         select case (char (parse_node_get_rule_key (pn_block)))
         case ("block_def")
            pn_block_spec => parse_node_get_sub_ptr (pn_block, 2)
```

1123

```
            pn_block_name => parse_node_get_sub_ptr (pn_block_spec)
            if (trim (adjustl (upper_case (parse_node_get_string &
                    (pn_block_name)))) == block_name) then
               return
            end if
         end select
         pn_block => parse_node_get_next_ptr (pn_block)
      end do
      if (required) then
         call msg_fatal ("SLHA: block '" // char (block_name) // "' not found")
      end if
    end function slha_get_block_ptr
```

Scan the file for the first/next DECAY block.

⟨*SLHA: procedures*⟩+≡

```
    function slha_get_first_decay_ptr (parse_tree) result (pn_decay)
      type(parse_node_t), pointer :: pn_decay
      type(parse_tree_t), intent(in) :: parse_tree
      type(parse_node_t), pointer :: pn_root
      pn_root => parse_tree_get_root_ptr (parse_tree)
      pn_decay => parse_node_get_sub_ptr (pn_root)
      do while (associated (pn_decay))
         select case (char (parse_node_get_rule_key (pn_decay)))
         case ("decay_def")
            return
         end select
         pn_decay => parse_node_get_next_ptr (pn_decay)
      end do
    end function slha_get_first_decay_ptr

    function slha_get_next_decay_ptr (pn_block) result (pn_decay)
      type(parse_node_t), pointer :: pn_decay
      type(parse_node_t), intent(in), target :: pn_block
      pn_decay => parse_node_get_next_ptr (pn_block)
      do while (associated (pn_decay))
         select case (char (parse_node_get_rule_key (pn_decay)))
         case ("decay_def")
            return
         end select
         pn_decay => parse_node_get_next_ptr (pn_decay)
      end do
    end function slha_get_next_decay_ptr
```

Extract and transfer data from blocks

Given the parse node of a block, find the parse node of a particular switch or
data line. Return this node and the node of the data item following the integer
code.

⟨*SLHA: procedures*⟩+≡

```
    subroutine slha_find_index_ptr (pn_block, pn_data, pn_item, code)
      type(parse_node_t), intent(in), target :: pn_block
  !   type(parse_node_t), intent(out), pointer :: pn_data
```

```
   !    type(parse_node_t), intent(out), pointer :: pn_item
        type(parse_node_t), pointer :: pn_data
        type(parse_node_t), pointer :: pn_item
        integer, intent(in) :: code
        pn_data => parse_node_get_sub_ptr (pn_block, 4)
        call slha_next_index_ptr (pn_data, pn_item, code)
      end subroutine slha_find_index_ptr

      subroutine slha_find_index_pair_ptr (pn_block, pn_data, pn_item, code1, code2)
        type(parse_node_t), intent(in), target :: pn_block
   !    type(parse_node_t), intent(out), pointer :: pn_data
   !    type(parse_node_t), intent(out), pointer :: pn_item
        type(parse_node_t), pointer :: pn_data
        type(parse_node_t), pointer :: pn_item
        integer, intent(in) :: code1, code2
        pn_data => parse_node_get_sub_ptr (pn_block, 4)
        call slha_next_index_pair_ptr (pn_data, pn_item, code1, code2)
      end subroutine slha_find_index_pair_ptr
```

Starting from the pointer to a data line, find a data line with the given integer
code.

⟨*SLHA: procedures*⟩+≡

```
    subroutine slha_next_index_ptr (pn_data, pn_item, code)
      type(parse_node_t), intent(inout), pointer :: pn_data
      integer, intent(in) :: code
  !    type(parse_node_t), intent(out), pointer :: pn_item
      type(parse_node_t), pointer :: pn_item
      type(parse_node_t), pointer :: pn_line, pn_code
      do while (associated (pn_data))
         pn_line => parse_node_get_sub_ptr (pn_data, 2)
         pn_code => parse_node_get_sub_ptr (pn_line)
         select case (char (parse_node_get_rule_key (pn_code)))
         case ("integer")
            if (parse_node_get_integer (pn_code) == code) then
               pn_item => parse_node_get_next_ptr (pn_code)
               return
            end if
         end select
         pn_data => parse_node_get_next_ptr (pn_data)
      end do
      pn_item => null ()
    end subroutine slha_next_index_ptr
```

Starting from the pointer to a data line, find a data line with the given integer
code pair.

⟨*SLHA: procedures*⟩+≡

```
    subroutine slha_next_index_pair_ptr (pn_data, pn_item, code1, code2)
      type(parse_node_t), intent(inout), pointer :: pn_data
      integer, intent(in) :: code1, code2
  !    type(parse_node_t), intent(out), pointer :: pn_item
      type(parse_node_t), pointer :: pn_item
      type(parse_node_t), pointer :: pn_line, pn_code1, pn_code2
      do while (associated (pn_data))
```

```
            pn_line => parse_node_get_sub_ptr (pn_data, 2)
            pn_code1 => parse_node_get_sub_ptr (pn_line)
            select case (char (parse_node_get_rule_key (pn_code1)))
            case ("integer")
               if (parse_node_get_integer (pn_code1) == code1) then
                  pn_code2 => parse_node_get_next_ptr (pn_code1)
                  if (associated (pn_code2)) then
                     select case (char (parse_node_get_rule_key (pn_code2)))
                     case ("integer")
                        if (parse_node_get_integer (pn_code2) == code2) then
                           pn_item => parse_node_get_next_ptr (pn_code2)
                           return
                        end if
                     end select
                  end if
               end if
            end select
            pn_data => parse_node_get_next_ptr (pn_data)
        end do
      pn_item => null ()
    end subroutine slha_next_index_pair_ptr
```

Handle info data

Return all strings with index i. The result is an allocated string array. Since
we do not know the number of matching entries in advance, we build an inter-
mediate list which is transferred to the final array and deleted before exiting.

⟨*SLHA: procedures*⟩+≡
```
    subroutine retrieve_strings_in_block (pn_block, code, str_array)
      type(parse_node_t), intent(in), target :: pn_block
      integer, intent(in) :: code
      type(string_t), dimension(:), allocatable, intent(out) :: str_array
      type(parse_node_t), pointer :: pn_data, pn_item
      type :: str_entry_t
         type(string_t) :: str
         type(str_entry_t), pointer :: next => null ()
      end type str_entry_t
      type(str_entry_t), pointer :: first => null ()
      type(str_entry_t), pointer :: current => null ()
      integer :: n
      n = 0
      call slha_find_index_ptr (pn_block, pn_data, pn_item, code)
      if (associated (pn_item)) then
         n = n + 1
         allocate (first)
         first%str = parse_node_get_string (pn_item)
         current => first
         do while (associated (pn_data))
            pn_data => parse_node_get_next_ptr (pn_data)
            call slha_next_index_ptr (pn_data, pn_item, code)
            if (associated (pn_item)) then
               n = n + 1
```

```
            allocate (current%next)
            current => current%next
            current%str = parse_node_get_string (pn_item)
         end if
      end do
      allocate (str_array (n))
      n = 0
      do while (associated (first))
         n = n + 1
         current => first
         str_array(n) = current%str
         first => first%next
         deallocate (current)
      end do
   else
      allocate (str_array (0))
   end if
 end subroutine retrieve_strings_in_block
```

Transfer data from SLHA to variables

Extract real parameter with index `i`. If it does not exist, retrieve it from the
variable list, using the given name.

⟨*SLHA: procedures*⟩+≡

```
 function get_parameter_in_block (pn_block, code, name, var_list) result (var)
   real(default) :: var
   type(parse_node_t), intent(in), target :: pn_block
   integer, intent(in) :: code
   type(string_t), intent(in) :: name
   type(var_list_t), intent(in), target :: var_list
   type(parse_node_t), pointer :: pn_data, pn_item
   call slha_find_index_ptr (pn_block, pn_data, pn_item, code)
   if (associated (pn_item)) then
      var = get_real_parameter (pn_item)
   else
      var = var_list_get_rval (var_list, name)
   end if
 end function get_parameter_in_block
```

Extract a real data item with index `i`. If it does exist, set it in the variable list,
using the given name. If the variable is not present in the variable list, ignore
it.

⟨*SLHA: procedures*⟩+≡

```
 subroutine set_data_item (pn_block, code, name, var_list)
   type(parse_node_t), intent(in), target :: pn_block
   integer, intent(in) :: code
   type(string_t), intent(in) :: name
   type(var_list_t), intent(inout), target :: var_list
   type(parse_node_t), pointer :: pn_data, pn_item
   call slha_find_index_ptr (pn_block, pn_data, pn_item, code)
   if (associated (pn_item)) then
      call var_list_set_real &
```

```
              (var_list, name,  get_real_parameter (pn_item), &
               is_known=.true., ignore=.true.)
       end if
    end subroutine set_data_item
```

Extract a real matrix element with index `i,j`. If it does exists, set it in the variable list, using the given name. If the variable is not present in the variable list, ignore it.

⟨*SLHA: procedures*⟩+≡
```
    subroutine set_matrix_element (pn_block, code1, code2, name, var_list)
      type(parse_node_t), intent(in), target :: pn_block
      integer, intent(in) :: code1, code2
      type(string_t), intent(in) :: name
      type(var_list_t), intent(inout), target :: var_list
      type(parse_node_t), pointer :: pn_data, pn_item
      call slha_find_index_pair_ptr (pn_block, pn_data, pn_item, code1, code2)
      if (associated (pn_item)) then
         call var_list_set_real &
               (var_list, name,  get_real_parameter (pn_item), &
                is_known=.true., ignore=.true.)
      end if
    end subroutine set_matrix_element
```

Transfer data from variables to SLHA

Get a real/integer parameter with index `i` from the variable list and write it to the current output file. In the integer case, we account for the fact that the variable is type real. If it does not exist, do nothing.

⟨*SLHA: procedures*⟩+≡
```
    subroutine write_integer_data_item (u, code, name, var_list, comment)
      integer, intent(in) :: u
      integer, intent(in) :: code
      type(string_t), intent(in) :: name
      type(var_list_t), intent(in) :: var_list
      character(*), intent(in) :: comment
      integer :: item
      if (var_list_exists (var_list, name)) then
         item = nint (var_list_get_rval (var_list, name))
         call write_integer_parameter (u, code, item, comment)
      end if
    end subroutine write_integer_data_item

    subroutine write_real_data_item (u, code, name, var_list, comment)
      integer, intent(in) :: u
      integer, intent(in) :: code
      type(string_t), intent(in) :: name
      type(var_list_t), intent(in) :: var_list
      character(*), intent(in) :: comment
      real(default) :: item
      if (var_list_exists (var_list, name)) then
         item = var_list_get_rval (var_list, name)
         call write_real_parameter (u, code, item, comment)
```

1128

```
      end if
    end subroutine write_real_data_item
```

Get a real data item with two integer indices from the variable list and write it to the current output file. If it does not exist, do nothing.

⟨*SLHA: procedures*⟩+≡
```
    subroutine write_matrix_element (u, code1, code2, name, var_list, comment)
      integer, intent(in) :: u
      integer, intent(in) :: code1, code2
      type(string_t), intent(in) :: name
      type(var_list_t), intent(in) :: var_list
      character(*), intent(in) :: comment
      real(default) :: item
      if (var_list_exists (var_list, name)) then
         item = var_list_get_rval (var_list, name)
         call write_real_matrix_element (u, code1, code2, item, comment)
      end if
    end subroutine write_matrix_element
```

15.0.4   Auxiliary function

Write a block header.

⟨*SLHA: procedures*⟩+≡
```
    subroutine write_block_header (u, name, comment)
      integer, intent(in) :: u
      character(*), intent(in) :: name, comment
      write (u, "(A,1x,A,3x,'#',1x,A)")  "BLOCK", name, comment
    end subroutine write_block_header
```

Extract a real parameter that may be defined real or integer, signed or unsigned.

⟨*SLHA: procedures*⟩+≡
```
    function get_real_parameter (pn_item) result (var)
      real(default) :: var
      type(parse_node_t), intent(in), target :: pn_item
      type(parse_node_t), pointer :: pn_sign, pn_var
      integer :: sign
      select case (char (parse_node_get_rule_key (pn_item)))
      case ("signed_number")
         pn_sign => parse_node_get_sub_ptr (pn_item)
         pn_var  => parse_node_get_next_ptr (pn_sign)
         select case (char (parse_node_get_key (pn_sign)))
         case ("+");  sign = +1
         case ("-");  sign = -1
         end select
      case default
         sign = +1
         pn_var => pn_item
      end select
      select case (char (parse_node_get_rule_key (pn_var)))
      case ("integer");  var = sign * parse_node_get_integer (pn_var)
      case ("real");     var = sign * parse_node_get_real (pn_var)
```

```
         end select
       end function get_real_parameter
```

Auxiliary: Extract an integer parameter that may be defined signed or unsigned.
A real value is an error.

⟨*SLHA: procedures*⟩+≡
```
    function get_integer_parameter (pn_item) result (var)
      integer :: var
      type(parse_node_t), intent(in), target :: pn_item
      type(parse_node_t), pointer :: pn_sign, pn_var
      integer :: sign
      select case (char (parse_node_get_rule_key (pn_item)))
      case ("signed_integer")
         pn_sign => parse_node_get_sub_ptr (pn_item)
         pn_var  => parse_node_get_next_ptr (pn_sign)
         select case (char (parse_node_get_key (pn_sign)))
         case ("+");  sign = +1
         case ("-");  sign = -1
         end select
      case ("integer")
         sign = +1
         pn_var => pn_item
      case default
         call parse_node_write (pn_var)
         call msg_error ("SLHA: Integer parameter expected")
         var = 0
         return
      end select
      var = sign * parse_node_get_integer (pn_var)
    end function get_integer_parameter
```

Write an integer parameter with a single index directly to file, using the required
output format.

⟨*SLHA: procedures*⟩+≡
```
    subroutine write_integer_parameter (u, code, item, comment)
      integer, intent(in) :: u
      integer, intent(in) :: code
      integer, intent(in) :: item
      character(*), intent(in) :: comment
 1    format (1x, I9, 3x, 3x, I9, 4x, 3x, '#', 1x, A)
      write (u, 1)  code, item, comment
    end subroutine write_integer_parameter
```

Write a real parameter with two indices directly to file, using the required output
format.

⟨*SLHA: procedures*⟩+≡
```
    subroutine write_real_parameter (u, code, item, comment)
      integer, intent(in) :: u
      integer, intent(in) :: code
      real(default), intent(in) :: item
      character(*), intent(in) :: comment
 1    format (1x, I9, 3x, 1P, E16.8, 0P, 3x, '#', 1x, A)
```

```
        write (u, 1)  code, item, comment
    end subroutine write_real_parameter
```

Write a real parameter with a single index directly to file, using the required
output format.

⟨*SLHA: procedures*⟩+≡
```
    subroutine write_real_matrix_element (u, code1, code2, item, comment)
      integer, intent(in) :: u
      integer, intent(in) :: code1, code2
      real(default), intent(in) :: item
      character(*), intent(in) :: comment
  1   format (1x, I2, 1x, I2, 3x, 1P, E16.8, 0P, 3x, '#', 1x, A)
      write (u, 1)  code1, code2, item, comment
    end subroutine write_real_matrix_element
```

The concrete SLHA interpreter

SLHA codes for particular physics models

⟨*SLHA: parameters*⟩+≡
```
    integer, parameter :: MDL_MSSM = 0
    integer, parameter :: MDL_NMSSM = 1
```

Take the parse tree and extract relevant data. Select the correct model and
store all data that is present in the appropriate variable list. Finally, update
the variable record.

⟨*SLHA: procedures*⟩+≡
```
    subroutine slha_interpret_parse_tree &
        (parse_tree, os_data, model, input, spectrum, decays)
      type(parse_tree_t), intent(in) :: parse_tree
      type(os_data_t), intent(in) :: os_data
  !   type(model_t), pointer, intent(out) :: model
      type(model_t), pointer, intent(inout) :: model
      logical, intent(in) :: input, spectrum, decays
      logical :: errors
      integer :: mssm_type
      call slha_handle_MODSEL (parse_tree, os_data, model, mssm_type)
      if (associated (model)) then
         if (input) then
            call slha_handle_SMINPUTS (parse_tree, model)
            call slha_handle_MINPAR (parse_tree, model, mssm_type)
         end if
         if (spectrum) then
            call slha_handle_info_block (parse_tree, "SPINFO", errors)
            if (errors)  return
            call slha_handle_MASS (parse_tree, model)
            call slha_handle_matrix_block (parse_tree, "NMIX", "mn_", 4, 4, model)
            call slha_handle_matrix_block (parse_tree, "NMNMIX", "mixn_", 5, 5, model)
            call slha_handle_matrix_block (parse_tree, "UMIX", "mu_", 2, 2, model)
            call slha_handle_matrix_block (parse_tree, "VMIX", "mv_", 2, 2, model)
            call slha_handle_matrix_block (parse_tree, "STOPMIX", "mt_", 2, 2, model)
            call slha_handle_matrix_block (parse_tree, "SBOTMIX", "mb_", 2, 2, model)
            call slha_handle_matrix_block (parse_tree, "STAUMIX", "ml_", 2, 2, model)
```

```
            call slha_handle_matrix_block (parse_tree, "NMHMIX", "mixh0_", 3, 3, model)
            call slha_handle_matrix_block (parse_tree, "NMAMIX", "mixa0_", 2, 3, model)
            call slha_handle_ALPHA (parse_tree, model)
            call slha_handle_HMIX (parse_tree, model)
            call slha_handle_NMSSMRUN (parse_tree, model)
            call slha_handle_matrix_block (parse_tree, "AU", "Au_", 3, 3, model)
            call slha_handle_matrix_block (parse_tree, "AD", "Ad_", 3, 3, model)
            call slha_handle_matrix_block (parse_tree, "AE", "Ae_", 3, 3, model)
         end if
         if (decays) then
            call slha_handle_info_block (parse_tree, "DCINFO", errors)
            if (errors)  return
            call slha_handle_decays (parse_tree, model)
         end if
      end if
   end subroutine slha_interpret_parse_tree
```

Info blocks

Handle the informational blocks SPINFO and DCINFO. The first two items are
program name and version. Items with index 3 are warnings. Items with index
4 are errors. We reproduce these as WHIZARD warnings and errors.

⟨*SLHA: procedures*⟩+≡

```
   subroutine slha_handle_info_block (parse_tree, block_name, errors)
     type(parse_tree_t), intent(in) :: parse_tree
     character(*), intent(in) :: block_name
     logical, intent(out) :: errors
     type(parse_node_t), pointer :: pn_block
     type(string_t), dimension(:), allocatable :: msg
     integer :: i
     pn_block => slha_get_block_ptr &
          (parse_tree, var_str (block_name), required=.true.)
     if (.not. associated (pn_block)) then
        call msg_error ("SLHA: Missing info block '" &
             // trim (block_name) // "'; ignored.")
        errors = .true.
        return
     end if
     select case (block_name)
     case ("SPINFO")
        call msg_message ("SLHA: SUSY spectrum program info:")
     case ("DCINFO")
        call msg_message ("SLHA: SUSY decay program info:")
     end select
     call retrieve_strings_in_block (pn_block, 1, msg)
     do i = 1, size (msg)
        call msg_message ("SLHA: " // char (msg(i)))
     end do
     call retrieve_strings_in_block (pn_block, 2, msg)
     do i = 1, size (msg)
        call msg_message ("SLHA: " // char (msg(i)))
     end do
```

```
      call retrieve_strings_in_block (pn_block, 3, msg)
      do i = 1, size (msg)
         call msg_warning ("SLHA: " // char (msg(i)))
      end do
      call retrieve_strings_in_block (pn_block, 4, msg)
      do i = 1, size (msg)
         call msg_error ("SLHA: " // char (msg(i)))
      end do
      errors = size (msg) > 0
   end subroutine slha_handle_info_block
```

MODSEL

Handle the overall model definition. Only certain models are recognized. The
soft-breaking model templates that determine the set of input parameters:

⟨*SLHA: parameters*⟩+≡
```
   integer, parameter :: MSSM_GENERIC = 0
   integer, parameter :: MSSM_SUGRA = 1
   integer, parameter :: MSSM_GMSB = 2
   integer, parameter :: MSSM_AMSB = 3
```

⟨*SLHA: procedures*⟩+≡
```
   subroutine slha_handle_MODSEL (parse_tree, os_data, model, mssm_type)
     type(parse_tree_t), intent(in) :: parse_tree
     type(os_data_t), intent(in) :: os_data
     type(model_t), pointer, intent(inout) :: model
     integer, intent(out) :: mssm_type
     type(parse_node_t), pointer :: pn_block, pn_data, pn_item
     type(string_t) :: model_name
     type(string_t) :: filename
     pn_block => slha_get_block_ptr &
          (parse_tree, var_str ("MODSEL"), required=.true.)
     call slha_find_index_ptr (pn_block, pn_data, pn_item, 1)
     if (associated (pn_item)) then
        mssm_type = get_integer_parameter (pn_item)
     else
        mssm_type = MSSM_GENERIC
     end if
     call slha_find_index_ptr (pn_block, pn_data, pn_item, 3)
     if (associated (pn_item)) then
        select case (parse_node_get_integer (pn_item))
        case (MDL_MSSM);  model_name = "MSSM"
        case (MDL_NMSSM); model_name = "NMSSM"
        case default
           call msg_fatal ("SLHA: unknown model code in MODSEL")
           return
        end select
     else
        model_name = "MSSM"
     end if
     call slha_find_index_ptr (pn_block, pn_data, pn_item, 4)
     if (associated (pn_item)) then
```

1133

```
          call msg_fatal (" R-parity violation is currently not supported by WHIZARD.")
        end if
        call slha_find_index_ptr (pn_block, pn_data, pn_item, 5)
        if (associated (pn_item)) then
          call msg_fatal (" CP violation is currently not supported by WHIZARD.")
        end if
        select case (char (model_name))
          case ("MSSM")
             select case (char (model_get_name (model)))
               case ("MSSM","MSSM_CKM","MSSM_Grav")
                  model_name = model_get_name (model)
               case default
                  call msg_fatal (" User-defined model and model in SLHA input file do not match.")
             end select
          case ("NMSSM")
             select case (char (model_get_name (model)))
               case ("NMSSM","NMSSM_CKM")
                  model_name = model_get_name (model)
               case default
                  call msg_fatal (" User-defined model and model in SLHA input file do not match.")
             end select
          case default
             call msg_fatal (" SLHA model selection failure.")
        end select
        filename = model_name // ".mdl"
        model => null ()
        call model_list_read_model (model_name, filename, os_data, model)
        if (associated (model)) then
           call msg_message ("SLHA: Initializing model '" &
                // char (model_name) // "'")
        else
           call msg_fatal ("SLHA: Initialization failed for model '" &
                // char (model_name) // "'")
        end if
    end subroutine slha_handle_MODSEL
```

Write a MODSEL block, based on the contents of the current model.

⟨*SLHA: procedures*⟩+≡

```
    subroutine slha_write_MODSEL (u, model, mssm_type)
      integer, intent(in) :: u
      type(model_t), intent(in), target :: model
      integer, intent(out) :: mssm_type
      type(var_list_t), pointer :: var_list
      integer :: model_id
      type(string_t) :: mtype_string
      var_list => model_get_var_list_ptr (model)
      if (var_list_exists (var_list, var_str ("mtype"))) then
         mssm_type = nint (var_list_get_rval (var_list, var_str ("mtype")))
      else
         call msg_error ("SLHA: parameter 'mtype' (SUSY breaking scheme) " &
              // "is unknown in current model, no SLHA output possible")
         mssm_type = -1
         return
```

```
        end if
        call write_block_header (u, "MODSEL", "SUSY model selection")
        select case (mssm_type)
        case (0);  mtype_string = "Generic MSSM"
        case (1);  mtype_string = "SUGRA"
        case (2);  mtype_string = "GMSB"
        case (3);  mtype_string = "AMSB"
        case default
           mtype_string = "unknown"
        end select
        call write_integer_parameter (u, 1, mssm_type, &
             "SUSY-breaking scheme: " // char (mtype_string))
        select case (char (model_get_name (model)))
        case ("MSSM");  model_id = MDL_MSSM
        case ("NMSSM"); model_id = MDL_NMSSM
        case default
           model_id = 0
        end select
        call write_integer_parameter (u, 3, model_id, &
             "SUSY model type: " // char (model_get_name (model)))
      end subroutine slha_write_MODSEL
```

SMINPUTS

Read SM parameters and update the variable list accordingly. If a parameter is not defined in the block, we use the previous value from the model variable list. For the basic parameters we have to do a small recalculation, since SLHA uses the G_F-α-m_Z scheme, while WHIZARD derives them from G_F, m_W, and m_Z.

⟨*SLHA: procedures*⟩+≡
```
    subroutine slha_handle_SMINPUTS (parse_tree, model)
      type(parse_tree_t), intent(in) :: parse_tree
      type(model_t), intent(inout), target :: model
      type(parse_node_t), pointer :: pn_block
      real(default) :: alpha_em_i, GF, alphas, mZ
      real(default) :: ee, vv, cw_sw, cw2, mW
      real(default) :: mb, mtop, mtau
      type(var_list_t), pointer :: var_list
      var_list => model_get_var_list_ptr (model)
      pn_block => slha_get_block_ptr &
           (parse_tree, var_str ("SMINPUTS"), required=.true.)
      if (.not. (associated (pn_block)))  return
      alpha_em_i = &
           get_parameter_in_block (pn_block, 1, var_str ("alpha_em_i"), var_list)
      GF = get_parameter_in_block (pn_block, 2, var_str ("GF"), var_list)
      alphas = &
           get_parameter_in_block (pn_block, 3, var_str ("alphas"), var_list)
      mZ   = get_parameter_in_block (pn_block, 4, var_str ("mZ"), var_list)
      mb   = get_parameter_in_block (pn_block, 5, var_str ("mb"), var_list)
      mtop = get_parameter_in_block (pn_block, 6, var_str ("mtop"), var_list)
      mtau = get_parameter_in_block (pn_block, 7, var_str ("mtau"), var_list)
      ee = sqrt (4 * pi / alpha_em_i)
      vv = 1 / sqrt (sqrt (2._default) * GF)
```

```
    cw_sw = ee * vv / (2 * mZ)
    if (2*cw_sw <= 1) then
       cw2 = (1 + sqrt (1 - 4 * cw_sw**2)) / 2
       mW = mZ * sqrt (cw2)
       call var_list_set_real (var_list, var_str ("GF"), GF, .true.)
       call var_list_set_real (var_list, var_str ("mZ"), mZ, .true.)
       call var_list_set_real (var_list, var_str ("mW"), mW, .true.)
       call var_list_set_real (var_list, var_str ("mtau"), mtau, .true.)
       call var_list_set_real (var_list, var_str ("mb"), mb, .true.)
       call var_list_set_real (var_list, var_str ("mtop"), mtop, .true.)
       call var_list_set_real (var_list, var_str ("alphas"), alphas, .true.)
    else
       call msg_fatal ("SLHA: Unphysical SM parameter values")
       return
    end if
  end subroutine slha_handle_SMINPUTS
```

Write a SMINPUTS block.

⟨*SLHA: procedures*⟩+≡
```
  subroutine slha_write_SMINPUTS (u, model)
    integer, intent(in) :: u
    type(model_t), intent(in), target :: model
    type(var_list_t), pointer :: var_list
    integer :: model_id
    var_list => model_get_var_list_ptr (model)
    call write_block_header (u, "SMINPUTS", "SM input parameters")
    call write_real_data_item (u, 1, var_str ("alpha_em_i"), var_list, &
         "Inverse electromagnetic coupling alpha (Z pole)")
    call write_real_data_item (u, 2, var_str ("GF"), var_list, &
         "Fermi constant")
    call write_real_data_item (u, 3, var_str ("alphas"), var_list, &
         "Strong coupling alpha_s (Z pole)")
    call write_real_data_item (u, 4, var_str ("mZ"), var_list, &
         "Z mass")
    call write_real_data_item (u, 5, var_str ("mb"), var_list, &
         "b running mass (at mb)")
    call write_real_data_item (u, 6, var_str ("mtop"), var_list, &
         "top mass")
    call write_real_data_item (u, 7, var_str ("mtau"), var_list, &
         "tau mass")
  end subroutine slha_write_SMINPUTS
```

MINPAR

The block of SUSY input parameters. They are accessible to WHIZARD, but
they only get used when an external spectrum generator is invoked. The precise
set of parameters depends on the type of SUSY breaking, which by itself is one
of the parameters.

⟨*SLHA: procedures*⟩+≡
```
  subroutine slha_handle_MINPAR (parse_tree, model, mssm_type)
    type(parse_tree_t), intent(in) :: parse_tree
    type(model_t), intent(inout), target :: model
```

```
      integer, intent(in) :: mssm_type
      type(var_list_t), pointer :: var_list
      type(parse_node_t), pointer :: pn_block
      var_list => model_get_var_list_ptr (model)
      call var_list_set_real (var_list, &
           var_str ("mtype"),  real(mssm_type, default), is_known=.true.)
      pn_block => slha_get_block_ptr &
           (parse_tree, var_str ("MINPAR"), required=.true.)
      select case (mssm_type)
      case (MSSM_SUGRA)
         call set_data_item (pn_block, 1, var_str ("m_zero"), var_list)
         call set_data_item (pn_block, 2, var_str ("m_half"), var_list)
         call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
         call set_data_item (pn_block, 4, var_str ("sgn_mu"), var_list)
         call set_data_item (pn_block, 5, var_str ("A0"), var_list)
      case (MSSM_GMSB)
         call set_data_item (pn_block, 1, var_str ("Lambda"), var_list)
         call set_data_item (pn_block, 2, var_str ("M_mes"), var_list)
         call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
         call set_data_item (pn_block, 4, var_str ("sgn_mu"), var_list)
         call set_data_item (pn_block, 5, var_str ("N_5"), var_list)
         call set_data_item (pn_block, 6, var_str ("c_grav"), var_list)
      case (MSSM_AMSB)
         call set_data_item (pn_block, 1, var_str ("m_zero"), var_list)
         call set_data_item (pn_block, 2, var_str ("m_grav"), var_list)
         call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
         call set_data_item (pn_block, 4, var_str ("sgn_mu"), var_list)
      case default
         call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
      end select
    end subroutine slha_handle_MINPAR
```

Write a MINPAR block as appropriate for the current model type.
⟨*SLHA: procedures*⟩+≡
```
    subroutine slha_write_MINPAR (u, model, mssm_type)
      integer, intent(in) :: u
      type(model_t), intent(in), target :: model
      integer, intent(in) :: mssm_type
      type(var_list_t), pointer :: var_list
      integer :: model_id
      var_list => model_get_var_list_ptr (model)
      call write_block_header (u, "MINPAR", "Basic SUSY input parameters")
      select case (mssm_type)
      case (MSSM_SUGRA)
         call write_real_data_item (u, 1, var_str ("m_zero"), var_list, &
              "Common scalar mass")
         call write_real_data_item (u, 2, var_str ("m_half"), var_list, &
              "Common gaugino mass")
         call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
              "tan(beta)")
         call write_integer_data_item (u, 4, &
              var_str ("sgn_mu"), var_list, &
              "Sign of mu")
         call write_real_data_item (u, 5, var_str ("A0"), var_list, &
```

```
                  "Common trilinear coupling")
        case (MSSM_GMSB)
           call write_real_data_item (u, 1, var_str ("Lambda"), var_list, &
                  "Soft-breaking scale")
           call write_real_data_item (u, 2, var_str ("M_mes"), var_list, &
                  "Messenger scale")
           call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
                  "tan(beta)")
           call write_integer_data_item (u, 4, &
                  var_str ("sgn_mu"), var_list, &
                  "Sign of mu")
           call write_integer_data_item (u, 5, var_str ("N_5"), var_list, &
                  "Messenger index")
           call write_real_data_item (u, 6, var_str ("c_grav"), var_list, &
                  "Gravitino mass factor")
        case (MSSM_AMSB)
           call write_real_data_item (u, 1, var_str ("m_zero"), var_list, &
                  "Common scalar mass")
           call write_real_data_item (u, 2, var_str ("m_grav"), var_list, &
                  "Gravitino mass")
           call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
                  "tan(beta)")
           call write_integer_data_item (u, 4, &
                  var_str ("sgn_mu"), var_list, &
                  "Sign of mu")
        case default
           call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
                  "tan(beta)")
        end select
      end subroutine slha_write_MINPAR
```

Mass spectrum

Set masses. Since the particles are identified by PDG code, read the line and try to set the appropriate particle mass in the current model. At the end, update parameters, just in case the W or Z mass was included.

⟨*SLHA: procedures*⟩+≡

```
      subroutine slha_handle_MASS (parse_tree, model)
        type(parse_tree_t), intent(in) :: parse_tree
        type(model_t), intent(inout), target :: model
        type(parse_node_t), pointer :: pn_block, pn_data, pn_line, pn_code
        type(parse_node_t), pointer :: pn_mass
        integer :: pdg
        real(default) :: mass
        pn_block => slha_get_block_ptr &
             (parse_tree, var_str ("MASS"), required=.true.)
        if (.not. (associated (pn_block)))  return
        pn_data => parse_node_get_sub_ptr (pn_block, 4)
        do while (associated (pn_data))
           pn_line => parse_node_get_sub_ptr (pn_data, 2)
           pn_code => parse_node_get_sub_ptr (pn_line)
           if (associated (pn_code)) then
```

```
        pdg = get_integer_parameter (pn_code)
        pn_mass => parse_node_get_next_ptr (pn_code)
        if (associated (pn_mass)) then
           mass = get_real_parameter (pn_mass)
           call model_set_particle_mass (model, pdg, mass)
        else
           call msg_error ("SLHA: Block MASS: Missing mass value")
        end if
     else
        call msg_error ("SLHA: Block MASS: Missing PDG code")
     end if
     pn_data => parse_node_get_next_ptr (pn_data)
   end do
 end subroutine slha_handle_MASS
```

Widths

Set widths. For each DECAY block, extract the header, read the PDG code
and width, and try to set the appropriate particle width in the current model.

⟨*SLHA: procedures*⟩+≡

```
 subroutine slha_handle_decays (parse_tree, model)
   type(parse_tree_t), intent(in) :: parse_tree
   type(model_t), intent(inout), target :: model
   type(parse_node_t), pointer :: pn_decay, pn_decay_spec, pn_code, pn_width
   integer :: pdg
   real(default) :: width
   pn_decay => slha_get_first_decay_ptr (parse_tree)
   do while (associated (pn_decay))
      pn_decay_spec => parse_node_get_sub_ptr (pn_decay, 2)
      pn_code => parse_node_get_sub_ptr (pn_decay_spec)
      pdg = get_integer_parameter (pn_code)
      pn_width => parse_node_get_next_ptr (pn_code)
      width = get_real_parameter (pn_width)
      call model_set_particle_width (model, pdg, width)
      pn_decay => slha_get_next_decay_ptr (pn_decay)
   end do
 end subroutine slha_handle_decays
```

Mixing matrices

Read mixing matrices. We can treat all matrices by a single procedure if we
just know the block name, variable prefix, and matrix dimension. The matrix
dimension must be less than 10. For the pseudoscalar Higgses in NMSSM-type
models we need off-diagonal matrices, so we generalize the definition.

⟨*SLHA: procedures*⟩+≡

```
 subroutine slha_handle_matrix_block &
      (parse_tree, block_name, var_prefix, dim1, dim2, model)
   type(parse_tree_t), intent(in) :: parse_tree
   character(*), intent(in) :: block_name, var_prefix
   integer, intent(in) :: dim1, dim2
   type(model_t), intent(inout), target :: model
```

```
      type(parse_node_t), pointer :: pn_block
      type(var_list_t), pointer :: var_list
      integer :: i, j
      character(len=len(var_prefix)+2) :: var_name
      var_list => model_get_var_list_ptr (model)
      pn_block => slha_get_block_ptr &
           (parse_tree, var_str (block_name), required=.false.)
      if (.not. (associated (pn_block)))  return
      do i = 1, dim1
         do j = 1, dim2
            write (var_name, "(A,I1,I1)")  var_prefix, i, j
            call set_matrix_element (pn_block, i, j, var_str (var_name), var_list)
         end do
      end do
    end subroutine slha_handle_matrix_block
```

Higgs data

Read the block ALPHA which holds just the Higgs mixing angle.

⟨*SLHA: procedures*⟩+≡

```
    subroutine slha_handle_ALPHA (parse_tree, model)
      type(parse_tree_t), intent(in) :: parse_tree
      type(model_t), intent(inout), target :: model
      type(parse_node_t), pointer :: pn_block, pn_line, pn_data, pn_item
      type(var_list_t), pointer :: var_list
      real(default) :: al_h
      var_list => model_get_var_list_ptr (model)
      pn_block => slha_get_block_ptr &
           (parse_tree, var_str ("ALPHA"), required=.false.)
      if (.not. (associated (pn_block)))  return
      pn_data => parse_node_get_sub_ptr (pn_block, 4)
      pn_line => parse_node_get_sub_ptr (pn_data, 2)
      pn_item => parse_node_get_sub_ptr (pn_line)
      if (associated (pn_item)) then
         al_h = get_real_parameter (pn_item)
         call var_list_set_real (var_list, var_str ("al_h"), al_h, &
              is_known=.true., ignore=.true.)
      end if
    end subroutine slha_handle_ALPHA
```

Read the block HMIX for the Higgs mixing parameters

⟨*SLHA: procedures*⟩+≡

```
    subroutine slha_handle_HMIX (parse_tree, model)
      type(parse_tree_t), intent(in) :: parse_tree
      type(model_t), intent(inout), target :: model
      type(parse_node_t), pointer :: pn_block
      type(var_list_t), pointer :: var_list
      var_list => model_get_var_list_ptr (model)
      pn_block => slha_get_block_ptr &
           (parse_tree, var_str ("HMIX"), required=.false.)
      if (.not. (associated (pn_block)))  return
      call set_data_item (pn_block, 1, var_str ("mu_h"), var_list)
```

```
      call set_data_item (pn_block, 2, var_str ("tanb_h"), var_list)
    end subroutine slha_handle_HMIX
```

Read the block NMSSMRUN for the specific NMSSM parameters

⟨*SLHA: procedures*⟩+≡
```
    subroutine slha_handle_NMSSMRUN (parse_tree, model)
      type(parse_tree_t), intent(in) :: parse_tree
      type(model_t), intent(inout), target :: model
      type(parse_node_t), pointer :: pn_block
      type(var_list_t), pointer :: var_list
      var_list => model_get_var_list_ptr (model)
      pn_block => slha_get_block_ptr &
            (parse_tree, var_str ("NMSSMRUN"), required=.false.)
      if (.not. (associated (pn_block)))  return
      call set_data_item (pn_block, 1, var_str ("ls"), var_list)
      call set_data_item (pn_block, 2, var_str ("ks"), var_list)
      call set_data_item (pn_block, 3, var_str ("a_ls"), var_list)
      call set_data_item (pn_block, 4, var_str ("a_ks"), var_list)
      call set_data_item (pn_block, 5, var_str ("nmu"), var_list)
    end subroutine slha_handle_NMSSMRUN
```

15.0.5   Parser

Read a SLHA file from stream, including preprocessing, and make up a parse tree.

⟨*SLHA: procedures*⟩+≡
```
    subroutine slha_parse_stream (stream, parse_tree)
      type(stream_t), intent(inout), target :: stream
      type(parse_tree_t), intent(out) :: parse_tree
      type(ifile_t) :: ifile
      type(lexer_t) :: lexer
      type(stream_t), target :: stream_tmp
      call slha_preprocess (stream, ifile)
      call stream_init (stream_tmp, ifile)
      call lexer_init_slha (lexer)
      call lexer_assign_stream (lexer, stream_tmp)
      call parse_tree_init (parse_tree, syntax_slha, lexer)
      call lexer_final (lexer)
      call stream_final (stream_tmp)
      call ifile_final (ifile)
    end subroutine slha_parse_stream
```

Read a SLHA file chosen by name. Check first the current directory, then the directory where SUSY input files should be located.

⟨*SLHA: procedures*⟩+≡
```
    subroutine slha_parse_file (file, os_data, parse_tree)
      type(string_t), intent(in) :: file
      type(os_data_t), intent(in) :: os_data
      type(parse_tree_t), intent(out) :: parse_tree
      logical :: exist
      type(string_t) :: filename
```

```
        type(stream_t), target :: stream
        call msg_message ("Reading SLHA input file '" // char (file) // "'")
        filename = file
        inquire (file=char(filename), exist=exist)
        if (.not. exist) then
           filename = os_data%whizard_susypath // "/" // file
           inquire (file=char(filename), exist=exist)
           if (.not. exist) then
              call msg_fatal ("SLHA input file '" // char (file) // "' not found")
              return
           end if
        end if
        call stream_init (stream, char (filename))
        call slha_parse_stream (stream, parse_tree)
        call stream_final (stream)
      end subroutine slha_parse_file
```

15.0.6 API

Read the SLHA file, parse it, and interpret the parse tree. The model parameters
retrieved from the file will be inserted into the appropriate model, which is
loaded and modified in the background. The pointer to this model is returned
as the last argument.

⟨*SLHA: public*⟩+≡
```
    public :: slha_read_file
```

⟨*SLHA: procedures*⟩+≡
```
    subroutine slha_read_file (file, os_data, model, input, spectrum, decays)
      type(string_t), intent(in) :: file
      type(os_data_t), intent(in) :: os_data
  !   type(model_t), pointer, intent(out) :: model
      type(model_t), pointer, intent(inout) :: model
      logical, intent(in) :: input, spectrum, decays
      type(parse_tree_t) :: parse_tree
      call slha_parse_file (file, os_data, parse_tree)
      if (associated (parse_tree_get_root_ptr (parse_tree))) then
         call slha_interpret_parse_tree &
              (parse_tree, os_data, model, input, spectrum, decays)
         call parse_tree_final (parse_tree)
         call model_parameters_update (model)
      end if
    end subroutine slha_read_file
```

Write the SLHA contents, as far as possible, to external file.

⟨*SLHA: public*⟩+≡
```
    public :: slha_write_file
```

⟨*SLHA: procedures*⟩+≡
```
    subroutine slha_write_file (file, model, input, spectrum, decays)
      type(string_t), intent(in) :: file
      type(model_t), target, intent(in) :: model
      logical, intent(in) :: input, spectrum, decays
```

```fortran
      integer :: mssm_type
      integer :: u
      u = free_unit ()
      call msg_message ("Writing SLHA output file '" // char (file) // "'")
      open (unit=u, file=char(file), action="write", status="replace")
      write (u, "(A)")  "# SUSY Les Houches Accord"
      write (u, "(A)")  "# Output generated by " // trim (VERSION_STRING)
      call slha_write_MODSEL (u, model, mssm_type)
      if (input) then
         call slha_write_SMINPUTS (u, model)
         call slha_write_MINPAR (u, model, mssm_type)
      end if
      if (spectrum) then
         call msg_bug ("SLHA: spectrum output not supported yet")
      end if
      if (decays) then
         call msg_bug ("SLHA: decays output not supported yet")
      end if
      close (u)
    end subroutine slha_write_file
```

15.0.7   Test

⟨*SLHA: public*⟩+≡
```fortran
    public :: slha_test
```

⟨*SLHA: procedures*⟩+≡
```fortran
    subroutine slha_test ()
      type(os_data_t) :: os_data
      type(parse_tree_t) :: parse_tree
      integer :: unit
      character(*), parameter :: file_test = "slha_test.out"
      character(*), parameter :: file_slha = "slha_test.dat"
      type(model_t), pointer :: model
      call os_data_init (os_data)
      call slha_parse_file (var_str ("sps1ap_decays.slha"), os_data, parse_tree)
      call msg_message ("Writing parse tree to '" // file_test // "'")
      unit = free_unit ()
      open (unit=unit, file=file_test, action="write", status="replace")
      call parse_tree_write (parse_tree, unit)
      call slha_interpret_parse_tree (parse_tree, os_data, model, &
           input=.true., spectrum=.true., decays=.true.)
      call parse_tree_final (parse_tree)
      call var_list_write (model_get_var_list_ptr (model), only_type=V_REAL)
      call msg_message ("Writing SLHA output to '" // file_slha // "'")
      call slha_write_file (var_str (file_slha), model, input=.true., &
           spectrum=.false., decays=.false.)
    end subroutine slha_test
```

1143

Chapter 16

Top level API

16.1 Commands

This module defines the command language of the main input file.

⟨commands.f90⟩≡
  ⟨*File header*⟩

```
module commands
```

  ⟨*Use kinds*⟩
```
    use kinds, only: double !NODEP!
```
  ⟨*Use strings*⟩
```
    use limits, only: ITERATIONS_DEFAULT_LIST_SIZE !NODEP!
    use constants !NODEP!
```
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use tao_random_numbers !NODEP!
    use md5
    use os_interface
   use ifiles
    use lexers
    use syntax_rules
    use parser
    use analysis
    use pdg_arrays
    use prt_lists
    use variables
    use expressions
    use models
    use state_matrices
    use flavors
    use quantum_numbers
    use polarizations
    use event_formats
    use hepmc_interface
    use stdhep_interface
    use beams
```

```
    use sf_isr
    use sf_epa
    use sf_ewa
    use sf_lhapdf
    use strfun
    use mappings
    use phs_forests
    use cascades
    use process_libraries
    use processes
    use decays
    use events
    use slha_interface
```

⟨*Standard module head*⟩

⟨*Commands: public*⟩

⟨*Commands: parameters*⟩

⟨*Commands: types*⟩

⟨*Commands: variables*⟩

```
contains
```

⟨*Commands: procedures*⟩

```
end module commands
```

16.1.1 Step lists for looping

An entry in the step list represents a triplet of initial value, final value, and step size. The latter could also be a factor, for multiplicative stepping. Alternatively, there may be only a single value, with no steps.

Each value is represented by an expression that can be evaluated.

⟨*Commands: parameters*⟩≡
```
    integer, parameter :: ST_NONE = 0
    integer, parameter :: ST_SINGLE = 1
    integer, parameter :: ST_LINEAR = 2
    integer, parameter :: ST_LOG = 3
```

⟨*Commands: types*⟩≡
```
    type :: step_spec_t
       private
       integer :: type = ST_NONE
       type(eval_tree_t) :: expr_beg
       type(eval_tree_t) :: expr_end
       type(eval_tree_t) :: expr_step
       type(step_spec_t), pointer :: next => null ()
    end type step_spec_t
```

⟨Commands: types⟩+≡

```
type :: step_list_t
   private
   type(step_spec_t), pointer :: first => null ()
   type(step_spec_t), pointer :: last => null ()
end type step_list_t
```

If the type is SINGLE, there is one parse node. For LINEAR and LOG, there are three.

⟨Commands: procedures⟩≡

```
subroutine step_list_append_val (step_list, type, var_list, pn1, pn2, pn3)
  type(step_list_t), intent(inout) :: step_list
  integer, intent(in) :: type
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), intent(in), target :: pn1
  type(parse_node_t), intent(in), optional, target :: pn2, pn3
  type(step_spec_t), pointer :: current
  allocate (current)
  current%type = type
  call eval_tree_init_expr (current%expr_beg, pn1, var_list)
  select case (type)
  case (ST_LINEAR, ST_LOG)
     call eval_tree_init_expr (current%expr_end,  pn2, var_list)
     call eval_tree_init_expr (current%expr_step, pn3, var_list)
  end select
  if (associated (step_list%last)) then
     step_list%last%next => current
  else
     step_list%first => current
  end if
  step_list%last => current
end subroutine step_list_append_val
```

Finalize.

⟨Commands: procedures⟩+≡

```
subroutine step_list_final (step_list)
  type(step_list_t), intent(inout) :: step_list
  type(step_spec_t), pointer :: current
  do while (associated (step_list%first))
     current => step_list%first
     step_list%first => current%next
     call eval_tree_final (current%expr_beg)
     call eval_tree_final (current%expr_end)
     call eval_tree_final (current%expr_step)
     deallocate (current)
  end do
  step_list%last => null ()
end subroutine step_list_final
```

16.1.2 Structure-function configuration

Mapping configuration

A mapping is defined for a particular set of x-parameters. It has a type and a set of real parameters whose meaning depends on the type.

⟨*Commands: types*⟩+≡

```
type :: sf_mapping_t
  private
  integer, dimension(:), allocatable :: index
  integer :: type = SFM_NONE
  real(default), dimension(:), allocatable :: par
end type sf_mapping_t
```

Output

⟨*Commands: procedures*⟩+≡

```
subroutine sf_mapping_write (sf_mapping, unit)
  type(sf_mapping_t), intent(in) :: sf_mapping
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A,I0,10(', #',I0))")  "Mapping for parameters #", &
       sf_mapping%index
  select case (sf_mapping%type)
  case (SFM_NONE);     write (u, "(3x,A)")  "[none]"
  case (SFM_PDFPAIR);  write (u, "(3x,A)")  "PDF pair mapping"
  case (SFM_ISRPAIR);  write (u, "(3x,A)")  "ISR pair mapping"
  case (SFM_EPAPAIR);  write (u, "(3x,A)")  "EPA pair mapping"
  case (SFM_EWAPAIR);  write (u, "(3x,A)")  "EWA pair mapping"
  end select
  if (allocated (sf_mapping%par)) then
     write (u, "(3x,A)", advance="no")  "Parameters = "
     write (u, *) sf_mapping%par
  end if
end subroutine sf_mapping_write
```

Single structure function

This type holds the configuration of a structure function or function pair.

⟨*Commands: types*⟩+≡

```
type :: sf_data_t
  private
  integer :: type = STRF_NONE
  logical, dimension(2) :: affects_beam = .false.
  integer :: n_parameters = 0
  type(lhapdf_data_t), dimension(2) :: lhapdf
  type(isr_data_t), dimension(2) :: isr
  type(epa_data_t), dimension(2) :: epa
  type(ewa_data_t), dimension(2) :: ewa
  logical :: has_mapping = .false.
  type(sf_mapping_t) :: mapping
  type(sf_data_t), pointer :: next => null ()
```

1147

```
      end type sf_data_t
```

Output:
⟨*Commands: procedures*⟩+≡
```
    subroutine sf_data_write (sf_data, unit)
      type(sf_data_t), intent(in) :: sf_data
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)")  "Structure function"
      do i = 1, 2
         if (sf_data%affects_beam(i)) then
            select case (sf_data%type)
            case (STRF_NONE)
               write (u, "(1x,A)") "[none]"
            case (STRF_LHAPDF)
               call lhapdf_data_write (sf_data%lhapdf(i), unit)
            case (STRF_ISR)
               call isr_data_write (sf_data%isr(i), unit)
            case (STRF_EPA)
               call epa_data_write (sf_data%epa(i), unit)
            case (STRF_EWA)
               call ewa_data_write (sf_data%ewa(i), unit)
            end select
         end if
      end do
      write (u, *)  "affects beams = ", sf_data%affects_beam
      write (u, *)  "n_parameters  = ", sf_data%n_parameters
      if (sf_data%has_mapping) then
         call sf_mapping_write (sf_data%mapping, unit)
      end if
    end subroutine sf_data_write
```

Initialize a dataset in the list with LHAPDF-specific options.

For the PDF pair case, we apply a mapping of the unit square which has a real-valued power as parameter. The default value is two.

⟨*Commands: procedures*⟩+≡
```
    subroutine sf_data_init_lhapdf &
         (sf_data, lhapdf_status, model, flv, file, member, photon_scheme)
      type(sf_data_t), intent(inout) :: sf_data
      type(lhapdf_status_t), intent(inout) :: lhapdf_status
      type(model_t), intent(in), target :: model
      type(flavor_t), dimension(2), intent(in) :: flv
      type(string_t), intent(in), optional :: file
      integer, intent(in), optional :: member
      integer, intent(in), optional :: photon_scheme
      integer :: i
      do i = 1, 2
         if (sf_data%affects_beam(i)) then
            call lhapdf_data_init (sf_data%lhapdf(i), lhapdf_status, &
                 model, flv(i), file, member, photon_scheme)
         end if
      end do
```

```
      if (all (sf_data%affects_beam)) then
         allocate (sf_data%mapping%index (2))
         sf_data%mapping%index = (/1, sf_data%n_parameters+1/)
         sf_data%mapping%type = SFM_PDFPAIR
         allocate (sf_data%mapping%par (1))
         sf_data%mapping%par = 2._default
         sf_data%has_mapping = .true.
      end if
    end subroutine sf_data_init_lhapdf
```

⟨*Commands: procedures*⟩+≡
```
    subroutine sf_data_init_isr &
        (sf_data, model, flv, alpha, q_max, mass, order)
      type(sf_data_t), intent(inout) :: sf_data
      type(model_t), intent(in), target :: model
      type(flavor_t), dimension(2), intent(in) :: flv
      real(default), intent(in) :: alpha, q_max
      real(default), intent(in), optional :: mass
      integer, intent(in), optional :: order
      integer :: i
      do i = 1, 2
         if (sf_data%affects_beam(i)) then
            call isr_data_init (sf_data%isr(i), &
                 model, flv(i), alpha, q_max, mass)
            if (present (order)) &
                 call isr_data_set_order (sf_data%isr(i), order)
            call isr_data_check (sf_data%isr(i))
         end if
      end do
 !      if (all (sf_data%affects_beam)) then
 !         allocate (sf_data%mapping%index (2))
 !         sf_data%mapping%index = (/1, sf_data%n_parameters+1/)
 !         sf_data%mapping%type = SFM_ISRPAIR
 !         allocate (sf_data%mapping%par (1))
 !         sf_data%mapping%par = 2._default
 !         sf_data%has_mapping = .true.
 !      end if
    end subroutine sf_data_init_isr
```

⟨*Commands: procedures*⟩+≡
```
    subroutine sf_data_init_epa &
        (sf_data, model, flv, alpha, x_min, q_min, E_max, mass)
      type(sf_data_t), intent(inout) :: sf_data
      type(model_t), intent(in), target :: model
      type(flavor_t), dimension(2), intent(in) :: flv
      real(default), intent(in) :: alpha, x_min, q_min, E_max
      real(default), intent(in), optional :: mass
      integer :: i
      do i = 1, 2
         if (sf_data%affects_beam(i)) then
            call epa_data_init (sf_data%epa(i), &
                 model, flv(i), alpha, x_min, q_min, E_max, mass)
            call epa_data_check (sf_data%epa(i))
```

```
        end if
     end do
     if (all (sf_data%affects_beam)) then
        allocate (sf_data%mapping%index (2))
        sf_data%mapping%index = (/1, sf_data%n_parameters+1/)
        sf_data%mapping%type = SFM_EPAPAIR
        allocate (sf_data%mapping%par (1))
        sf_data%mapping%par = 1._default
        sf_data%has_mapping = .true.
     end if
  end subroutine sf_data_init_epa
```

⟨*Commands: procedures*⟩+≡

```
  subroutine sf_data_init_ewa &
       (sf_data, model, flv, x_min, q_min, pt_max, sqrts, &
        keep_momentum, keep_energy, mass)
    type(sf_data_t), intent(inout) :: sf_data
    type(model_t), intent(in), target :: model
    type(flavor_t), dimension(2), intent(in) :: flv
    real(default), intent(in) ::  x_min, q_min, pt_max, sqrts
    logical, intent(in) :: keep_momentum, keep_energy
    real(default), intent(in), optional :: mass
    integer :: i
    do i = 1, 2
       if (sf_data%affects_beam(i)) then
          call ewa_data_init (sf_data%ewa(i), &
               model, flv(i), x_min, q_min, pt_max, sqrts, &
               keep_momentum, keep_energy, mass)
          call ewa_data_check (sf_data%ewa(i))
       end if
    end do
    if (all (sf_data%affects_beam)) then
       allocate (sf_data%mapping%index (2))
       sf_data%mapping%index = (/1, sf_data%n_parameters+1/)
       sf_data%mapping%type = SFM_EWAPAIR
       allocate (sf_data%mapping%par (1))
       sf_data%mapping%par = 1._default
       sf_data%has_mapping = .true.
    end if
    if (keep_momentum .or. keep_energy) then
       sf_data%n_parameters = 3
    else
       sf_data%n_parameters = 1
    end if
  end subroutine sf_data_init_ewa
```

Structure function list

A list of structure functions and associated mappings.

⟨*Commands: types*⟩+≡

```
  type :: sf_list_t
     private
```

```
    integer :: n_strfun = 0
    integer :: n_mapping = 0
    type(sf_data_t), pointer :: first => null ()
    type(sf_data_t), pointer :: last => null ()
    character(32) :: md5sum = ""
  end type sf_list_t
```

Output

⟨*Commands: procedures*⟩+≡
```
  subroutine sf_list_write (sf_list, unit)
    type(sf_list_t), intent(in) :: sf_list
    integer, intent(in), optional :: unit
    integer :: u
    type(sf_data_t), pointer :: current
    u = output_unit (unit);  if (u < 0)  return
    write (u, "(A)")  "Structure function list"
    if (associated (sf_list%first)) then
       current => sf_list%first
       do while (associated (current))
          call sf_data_write (current, unit)
          current => current%next
       end do
    else
       write (u, "(1x,A)") "[empty]"
    end if
  end subroutine sf_list_write
```

Append a data set to the list. For EPA, several data sets may be used in parallel.
Allocate only one.

⟨*Commands: procedures*⟩+≡
```
  subroutine sf_list_append (sf_list, type, affects_beam, n_parameters, current)
    type(sf_list_t), intent(inout) :: sf_list
    integer, intent(in) :: type
    logical, dimension(2), intent(in) :: affects_beam
    integer, intent(in) :: n_parameters
    type(sf_data_t), pointer :: current
    allocate (current)
    current%type = type
    current%affects_beam = affects_beam
    current%n_parameters = n_parameters
    if (associated (sf_list%last)) then
       sf_list%last%next => current
    else
       sf_list%first => current
    end if
    sf_list%last => current
    sf_list%n_strfun = sf_list%n_strfun + count (affects_beam)
  end subroutine sf_list_append
```

Count mappings.

⟨*Commands: procedures*⟩+≡
```
  subroutine sf_list_freeze (sf_list)
```

```
      type(sf_list_t), intent(inout) :: sf_list
      type(sf_data_t), pointer :: current
      sf_list%n_mapping = 0
      current => sf_list%first
      do while (associated (current))
         if (current%has_mapping) then
            sf_list%n_mapping = sf_list%n_mapping + 1
         end if
         current => current%next
      end do
    end subroutine sf_list_freeze
```

Finalize.

⟨Commands: procedures⟩+≡
```
    subroutine sf_list_final (sf_list)
      type(sf_list_t), intent(inout) :: sf_list
      type(sf_data_t), pointer :: current
      do while (associated (sf_list%first))
         current => sf_list%first
         sf_list%first => sf_list%first%next
         deallocate (current)
      end do
      sf_list%last => null ()
      sf_list%n_strfun = 0
    end subroutine sf_list_final
```

Return number of structure functions.

⟨Commands: procedures⟩+≡
```
    function sf_list_get_n_strfun (sf_list) result (n)
      integer :: n
      type(sf_list_t), intent(in) :: sf_list
      n = sf_list%n_strfun
    end function sf_list_get_n_strfun
```

Return number of mappings.

⟨Commands: procedures⟩+≡
```
    function sf_list_get_n_mapping (sf_list) result (n)
      integer :: n
      type(sf_list_t), intent(in) :: sf_list
      n = sf_list%n_mapping
    end function sf_list_get_n_mapping
```

Return the MD5 checksum.

⟨Commands: procedures⟩+≡
```
    function sf_list_get_md5sum (sf_list) result (sf_md5sum)
      character(32) :: sf_md5sum
      type(sf_list_t), intent(in) :: sf_list
      sf_md5sum = sf_list%md5sum
    end function sf_list_get_md5sum
```

Compute the MD5 checksum.

⟨*Commands: procedures*⟩+≡

```
subroutine sf_list_compute_md5sum (sf_list)
  type(sf_list_t), intent(inout) :: sf_list
  integer :: unit
  unit = free_unit ()
  open (unit = unit, status = "scratch", action = "readwrite")
  call sf_list_write (sf_list, unit)
  rewind (unit)
  sf_list%md5sum = md5sum (unit)
  close (unit)
end subroutine sf_list_compute_md5sum
```

Transfer to the process object

Initialize actual structure functions, once the `sf_list` is complete. Beam initialization has to come before this.

⟨*Commands: procedures*⟩+≡

```
subroutine process_setup_strfun (process, sf_list)
  type(process_t), intent(inout), target :: process
  type(sf_list_t), intent(in) :: sf_list
  type(sf_data_t), pointer :: current
  integer :: i_sf, j, i_map, i_par
  i_sf = 0
  i_map = 0
  i_par = 0
  current => sf_list%first
  do while (associated (current))
     if (current%has_mapping) then
        i_map = i_map + 1
        call process_set_strfun_mapping &
             (process, i_map, i_par + current%mapping%index, &
              current%mapping%type, current%mapping%par)
     end if
     do j = 1, 2
        if (current%affects_beam(j)) then
           i_sf = i_sf + 1
           select case (current%type)
           case (STRF_LHAPDF)
              call process_set_strfun &
                   (process, i_sf, j, current%lhapdf(j), current%n_parameters)
           case (STRF_ISR)
              call process_set_strfun &
                   (process, i_sf, j, current%isr(j), current%n_parameters)
           case (STRF_EPA)
              call process_set_strfun &
                   (process, i_sf, j, current%epa(j), current%n_parameters)
           case (STRF_EWA)
              call process_set_strfun &
                   (process, i_sf, j, current%ewa(j), current%n_parameters)
           end select
           i_par = i_par + current%n_parameters
```

```
          end if
        end do
        current => current%next
      end do
    end subroutine process_setup_strfun
```

16.1.3 Integration passes

Each integration pass has a number of iterations and a number of calls per iteration. The last pass produces the end result; the previous passes are used for adaptation.

TODO: Allow for more options.

⟨*Commands: types*⟩+≡
```
    type :: iterations_spec_t
      private
      integer :: n_it = 0
      integer :: n_calls = 0
    end type iterations_spec_t
```

We build up a list of iterations.

⟨*Commands: types*⟩+≡
```
    type :: iterations_list_t
      private
      integer :: n_pass = 0
      type(iterations_spec_t), dimension(:), allocatable :: pass
    end type iterations_list_t
```

⟨*Commands: procedures*⟩+≡
```
    subroutine iterations_list_init (it_list, n_it, n_calls)
      type(iterations_list_t), intent(inout) :: it_list
      integer, dimension(:), intent(in) :: n_it, n_calls
      it_list%n_pass = size (n_it)
      if (allocated (it_list%pass)) deallocate (it_list%pass)
      allocate (it_list%pass (it_list%n_pass))
      it_list%pass%n_it = n_it
      it_list%pass%n_calls = n_calls
    end subroutine iterations_list_init
```

Fill all zero entries by corresponding entries from a default iterations list:

⟨*Commands: procedures*⟩+≡
```
    subroutine iterations_list_complete (it_list, it_list_default)
      type(iterations_list_t), intent(inout) :: it_list
      type(iterations_list_t), intent(in) :: it_list_default
      if (it_list%n_pass >= 1) then
         if (it_list%pass(1)%n_it == 0)  &
              it_list%pass(1)%n_it = it_list_default%pass(1)%n_it
         if (it_list%pass(1)%n_calls == 0)  &
              it_list%pass(1)%n_calls = it_list_default%pass(1)%n_calls
      end if
      if (it_list%n_pass >= 2) then
```

```
          where (it_list%pass%n_it == 0) &
               it_list%pass%n_it = it_list_default%pass(2)%n_it
          where (it_list%pass%n_calls == 0) &
               it_list%pass%n_calls = it_list_default%pass(2)%n_calls
       end if
    end subroutine iterations_list_complete
```

⟨*Commands: procedures*⟩+≡
```
    subroutine iterations_list_clear (it_list)
      type(iterations_list_t), intent(inout) :: it_list
      it_list%n_pass = 0
      deallocate (it_list%pass)
    end subroutine iterations_list_clear
```

Write the list of iterations as a message.

⟨*Commands: procedures*⟩+≡
```
    subroutine iterations_list_write (it_list, unit)
      type(iterations_list_t), intent(in) :: it_list
      integer, intent(in), optional :: unit
      type(string_t) :: buffer
      character(30) :: ibuf
      integer :: i
      buffer = "iterations = "
      if (it_list%n_pass > 0) then
         do i = 1, it_list%n_pass
            if (i > 1)  buffer = buffer // ", "
            write (ibuf, "(I0,':',I0)") &
                  it_list%pass(i)%n_it, it_list%pass(i)%n_calls
            buffer = buffer // trim (ibuf)
         end do
      else
         buffer = buffer // "[undefined]"
      end if
      call msg_message (char (buffer), unit)
    end subroutine iterations_list_write
```

Transform the iterations list into arrays that indicate pass index and number of
calls for each iteration.

⟨*Commands: procedures*⟩+≡
```
    function iterations_list_get_pass_array (it_list) result (pass)
      integer, dimension(:), allocatable :: pass
      type(iterations_list_t), intent(in) :: it_list
      integer :: it, i
      allocate (pass (sum (it_list%pass%n_it)))
      it = 0
      do i = 1, it_list%n_pass
         pass(it+1 : it+it_list%pass(i)%n_it) = i
         it = it + it_list%pass(i)%n_it
      end do
    end function iterations_list_get_pass_array

    function iterations_list_get_n_calls_array (it_list) result (n_calls)
```

```
    integer, dimension(:), allocatable :: n_calls
    type(iterations_list_t), intent(in) :: it_list
    integer :: it, i
    allocate (n_calls (sum (it_list%pass%n_it)))
    it = 0
    do i = 1, it_list%n_pass
       n_calls(it+1 : it+it_list%pass(i)%n_it) = it_list%pass(i)%n_calls
       it = it + it_list%pass(i)%n_it
    end do
  end function iterations_list_get_n_calls_array
```

Return the total number of iterations

⟨*Commands: procedures*⟩+≡
```
  function iterations_list_get_n_it (it_list) result (n_it)
    integer :: n_it
    type(iterations_list_t), intent(in) :: it_list
    n_it = sum (it_list%pass%n_it)
  end function iterations_list_get_n_it
```

This routine corrects for a number of calls that is too low.

⟨*Commands: procedures*⟩+≡
```
  subroutine iterations_list_adjust_n_calls (it_list, process, grid_parameters)
    type(iterations_list_t), intent(inout), target :: it_list
    type(process_t), intent(in) :: process
    type(grid_parameters_t), intent(in) :: grid_parameters
    type(iterations_spec_t), pointer :: it_spec
    integer :: n_calls, pass
    logical :: changed
    changed = .false.
    do pass = 1, it_list%n_pass
       it_spec => it_list%pass(pass)
       n_calls = max (it_spec%n_calls, &
            process_get_n_channels (process) &
            * grid_parameters%min_calls_per_channel)
       if (n_calls /= it_spec%n_calls) then
          it_spec%n_calls = n_calls
          changed = .true.
       end if
    end do
    if (changed) then
       write (msg_buffer, "(A,I0)") "Process '" &
            // char (process_get_id (process)) // "': " &
            // "resetting n_calls to ", n_calls
       call msg_warning ()
    end if
  end subroutine iterations_list_adjust_n_calls
```

Allocate the default iterations lists and fill with some sensible values. This is implemented as a pointer since it is not intended to be modified by the user; the local RT dataset will thus contain another pointer to the same list, not a copy.

⟨*Limits: public parameters*⟩+≡

```
    integer, parameter, public :: ITERATIONS_DEFAULT_LIST_SIZE = 7
```
⟨*Commands: procedures*⟩+≡
```
    subroutine iterations_lists_init_default (it_list)
      type(iterations_list_t), dimension(:), pointer :: it_list
      allocate (it_list (ITERATIONS_DEFAULT_LIST_SIZE))
      call iterations_list_init (it_list(2), (/  3, 3 /), (/   1000,  10000 /))
      call iterations_list_init (it_list(3), (/  5, 3 /), (/   5000,  10000 /))
      call iterations_list_init (it_list(4), (/ 10, 5 /), (/  10000,  20000 /))
      call iterations_list_init (it_list(5), (/ 10, 5 /), (/  20000,  50000 /))
      call iterations_list_init (it_list(6), (/ 15, 5 /), (/  50000, 100000 /))
      call iterations_list_init (it_list(7), (/ 20, 5 /), (/  50000, 200000 /))
    end subroutine iterations_lists_init_default
```

16.1.4   File configuration

Files have a name and a format; we need lists of file specifications. Writing
LHEF files, we need beam information and overall process data.

⟨*Commands: parameters*⟩+≡
```
    integer, parameter :: FMT_NONE = 0
    integer, parameter :: FMT_DEFAULT = 1
    integer, parameter :: FMT_DEBUG = 2
    integer, parameter :: FMT_HEPMC = 10
    integer, parameter :: FMT_LHEF = 20
    integer, parameter :: FMT_LHA = 21
    integer, parameter :: FMT_HEPEVT = 30
    integer, parameter :: FMT_ASCII_SHORT = 31
    integer, parameter :: FMT_ASCII_LONG = 32
    integer, parameter :: FMT_ATHENA = 33
    integer, parameter :: FMT_STDHEP = 40
    integer, parameter :: FMT_STDHEP_UP = 41
```
⟨*Commands: types*⟩+≡
```
    type :: file_spec_t
       private
       type(string_t) :: name
       integer :: format = FMT_NONE
       type(hepmc_iostream_t), pointer :: iostream => null ()
       integer :: unit = 0
       type(flavor_t), dimension(:), allocatable :: beam_flv
       real(default), dimension(:), allocatable :: beam_energy
       real(default), dimension(:), allocatable :: integral
       real(default), dimension(:), allocatable :: error
       integer :: n_processes = 0
       logical :: unweighted = .true.
       logical :: negative_weights = .false.
       logical :: keep_beams = .false.
       type(file_spec_t), pointer :: next => null ()
    end type file_spec_t
```

File lists.

⟨*Commands: types*⟩+≡
```
    type :: file_list_t
```

```
      private
      type(file_spec_t), pointer :: first => null ()
      type(file_spec_t), pointer :: last => null ()
    end type file_list_t
```

⟨*Commands: procedures*⟩+≡

```
    subroutine file_list_append_file_spec &
        (file_list, procname, var_list, format, beam_flv, beam_energy, &
         n_processes) ! unweighted, negative_weights, &
      type(file_list_t), intent(inout) :: file_list
      type(var_list_t), intent(in) :: var_list
      integer, intent(in) :: format
      type(flavor_t), dimension(:), intent(in) :: beam_flv
      real(default), dimension(:), intent(in) :: beam_energy
      integer, intent(in) :: n_processes
!     logical, intent(in) :: unweighted, negative_weights
      type(string_t) :: basename
      type(string_t), intent(in) :: procname
      type(file_spec_t), pointer :: current
      basename = var_list_get_sval (var_list, var_str ("$sample"))
      if (basename == "") basename = procname
      allocate (current)
!     if (name /= "") then
!        current%name = name
!     else
        select case (format)
        case (FMT_DEFAULT);     current%name = basename // "." // var_list_get_sval &
              (var_list, var_str ("$extension_default"))
        case (FMT_DEBUG);       current%name = basename // "." // var_list_get_sval &
              (var_list, var_str ("$extension_debug"))
        case (FMT_HEPMC);       current%name = basename // "." // var_list_get_sval &
              (var_list, var_str ("$extension_hepmc"))
        case (FMT_LHEF);        current%name = basename // "." // var_list_get_sval &
              (var_list, var_str ("$extension_lhef"))
        case (FMT_LHA);         current%name = basename // "." // var_list_get_sval &
              (var_list, var_str ("$extension_lha"))
        case (FMT_HEPEVT);      current%name = basename // "." // var_list_get_sval &
              (var_list, var_str ("$extension_hepevt"))
        case (FMT_ASCII_SHORT); current%name = basename // "." // var_list_get_sval &
              (var_list, var_str ("$extension_ascii_short"))
        case (FMT_ASCII_LONG);  current%name = basename // "." // var_list_get_sval &
              (var_list, var_str ("$extension_ascii_long"))
        case (FMT_ATHENA);      current%name = basename // "." // var_list_get_sval &
              (var_list, var_str ("$extension_athena"))
        case (FMT_STDHEP);      current%name = basename // "." // var_list_get_sval &
              (var_list, var_str ("$extension_stdhep"))
        case (FMT_STDHEP_UP);   current%name = basename // "." // var_list_get_sval &
              (var_list, var_str ("$extension_stdhep_up"))
        case default;           current%name = basename // "." // var_list_get_sval &
              (var_list, var_str ("$extension_default"))
        end select
!     end if
      current%format = format
      allocate (current%beam_flv (size (beam_flv)))
```

```
         current%beam_flv = beam_flv
         allocate (current%beam_energy (size (beam_energy)))
         current%beam_energy = beam_energy
         current%n_processes = n_processes
         current%keep_beams = var_list_get_lval (var_list, var_str ("?keep_beams"))
         if (associated (file_list%last)) then
            file_list%last%next => current
         else
            file_list%first => current
         end if
         file_list%last => current
      end subroutine file_list_append_file_spec
```

⟨*Commands: procedures*⟩+≡
```
      subroutine file_list_final (file_list)
        type(file_list_t), intent(inout) :: file_list
        type(file_spec_t), pointer :: current
        do while (associated (file_list%first))
           current => file_list%first
           file_list%first => current%next
           deallocate (current)
        end do
        file_list%last => null ()
      end subroutine file_list_final
```

Open appropriate event files.
    LHEF: Initialize run data with beam and simulation parameters.

⟨*Commands: procedures*⟩+≡
```
      subroutine file_list_open (file_list, simulate)
        type(file_list_t), intent(inout), target :: file_list
        type(cmd_simulate_t), intent(inout), target :: simulate
        real(default), dimension(:), allocatable :: integral, error
        type(process_t), pointer :: process
        type(file_spec_t), pointer :: current
        integer :: i
        integer(i64) :: n_events_expected
        current => file_list%first
        allocate (integral (simulate%n_proc), error (simulate%n_proc))
        do i = 1, simulate%n_proc
           process => process_store_get_process_ptr ( &
                 simulate%process_id(i))
           if (associated (process)) then
              integral(i) = process_get_integral (process)
              error(i)    = process_get_error (process)
           else
              integral(i) = 0
              error(i) = 0
           end if
        end do
        n_events_expected = simulate%n_evt
        do while (associated (current))
           select case (current%format)
           case (FMT_DEFAULT)
```

```
      call msg_message ("Writing events in human-readable format " &
            // "to file '" // char (current%name) // "'")
      current%unit = free_unit ()
      open (unit=current%unit, file=char(current%name), &
            action="write", status="replace")
   case (FMT_DEBUG)
      call msg_message ("Writing events in verbose format to file '" &
            // char (current%name) // "'")
      current%unit = free_unit ()
      open (unit=current%unit, file=char(current%name), &
            action="write", status="replace")
   case (FMT_HEPMC)
      call msg_message ("Writing events in HepMC format to file '" &
            // char (current%name) // "'")
      if (hepmc_is_available ()) then
         allocate (current%iostream)
         call hepmc_iostream_open_out (current%iostream, current%name)
      else
         call msg_error ("HepMC event writing is disabled " &
               // "because HepMC library is not linked.")
      end if
   case (FMT_HEPEVT)
      call msg_message ("Writing events in HEPEVT format to file '" &
            // char (current%name) // "'")
      current%unit = free_unit ()
      open (unit=current%unit, file=char(current%name), &
            action="write", status="replace")
   case (FMT_ASCII_SHORT)
      call msg_message ("Writing events in short ASCII format to file '" &
            // char (current%name) // "'")
      current%unit = free_unit ()
      open (unit=current%unit, file=char(current%name), &
            action="write", status="replace")
   case (FMT_ASCII_LONG)
      call msg_message ("Writing events in long ASCII format to file '" &
            // char (current%name) // "'")
      current%unit = free_unit ()
      open (unit=current%unit, file=char(current%name), &
            action="write", status="replace")
   case (FMT_ATHENA)
      call msg_message ("Writing events in ATHENA format to file '" &
            // char (current%name) // "'")
      current%unit = free_unit ()
      open (unit=current%unit, file=char(current%name), &
            action="write", status="replace")
   case (FMT_LHEF)
      call msg_message ("Writing events in LHEF format to file '" &
            // char (current%name) // "'")
      current%unit = free_unit ()
      open (unit=current%unit, file=char(current%name), &
            action="write", status="replace")
      call les_houches_events_write_header (current%unit)
      call heprup_init &
            (flavor_get_pdg (current%beam_flv), &
```

```
                   current%beam_energy, &
                   n_processes = current%n_processes, &
                   unweighted = current%unweighted, &
                   negative_weights = current%negative_weights)
              do i = 1, simulate%n_proc
                 call heprup_set_process_parameters (i = i, process_id = &
                    i, cross_section = integral(i), error = error(i))
              end do
              call heprup_write_lhef (current%unit)
           case (FMT_LHA)
              call msg_message ("Writing events in (old) LHA format to file '" &
                   // char (current%name) // "'")
              current%unit = free_unit ()
              open (unit=current%unit, file=char(current%name), &
                   action="write", status="replace")
              call heprup_init &
                   (flavor_get_pdg (current%beam_flv), &
                    current%beam_energy, &
                    n_processes = current%n_processes, &
                    unweighted = current%unweighted, &
                    negative_weights = current%negative_weights)
              do i = 1, simulate%n_proc
                 call heprup_set_process_parameters (i = i, process_id = &
                    i, cross_section = integral(i), error = error(i))
              end do
           case (FMT_STDHEP)
              call msg_message ("Writing events in binary STDHEP/HEPEVT format to file '" &
                   // char (current%name) // "'")
              call stdhep_init (char(current%name), "WHIZARD event sample", &
                   n_events_expected)
           case (FMT_STDHEP_UP)
              call msg_message ("Writing events in binary STDHEP/HEPRUP/HEPEUP format to file '" &
                   // char (current%name) // "'")
              call heprup_init &
                   (flavor_get_pdg (current%beam_flv), &
                    current%beam_energy, &
                    n_processes = current%n_processes, &
                    unweighted = current%unweighted, &
                    negative_weights = current%negative_weights)
              do i = 1, simulate%n_proc
                 call heprup_set_process_parameters (i = i, process_id = &
                    i, cross_section = integral(i), error = error(i))
              end do
              call stdhep_init (char(current%name), "WHIZARD event sample", &
                   n_events_expected)
              call stdhep_write (STDHEP_HEPRUP)
           end select
           current => current%next
        end do
     end subroutine file_list_open
```

Scan the file list and write the event in the selected formats.

⟨*Commands: procedures*⟩+≡
```
     subroutine file_list_write_event (file_list, event, i_proc, i_evt)
```

```
      type(file_list_t), intent(in), target :: file_list
      type(event_t), intent(in), target :: event
      integer, intent(in), optional :: i_proc, i_evt
      type(file_spec_t), pointer :: current
      type(hepmc_event_t) :: hepmc_event
      current => file_list%first
      do while (associated (current))
         select case (current%format)
         case (FMT_DEFAULT)
            call event_write (event, current%unit, verbose=.false.)
         case (FMT_DEBUG)
            call event_write (event, current%unit, verbose=.true.)
         case (FMT_HEPMC)
            if (hepmc_is_available ()) then
               call hepmc_event_init (hepmc_event, i_proc, i_evt)
               call event_write_to_hepmc (event, hepmc_event)
               call hepmc_iostream_write_event (current%iostream, hepmc_event)
               call hepmc_event_final (hepmc_event)
            end if
         case (FMT_HEPEVT)
            call event_write_to_hepevt (event, current%keep_beams)
            call hepevt_write_hepevt (current%unit)
         case (FMT_ASCII_SHORT)
            call event_write_to_hepevt (event, current%keep_beams)
            call hepevt_write_ascii (current%unit, .false.)
         case (FMT_ASCII_LONG)
            call event_write_to_hepevt (event, current%keep_beams)
            call hepevt_write_ascii (current%unit, .true.)
         case (FMT_ATHENA)
            call event_write_to_hepevt (event, current%keep_beams)
            call hepevt_write_athena (unit=current%unit, i_evt=i_evt)
         case (FMT_LHEF)
            call event_write_to_hepeup (event)
            call hepeup_write_lhef (current%unit)
         case (FMT_LHA)
            call event_write_to_hepeup (event)
            call hepeup_write_lha (current%unit)
         case (FMT_STDHEP)
            call event_write_to_hepevt (event, i_evt=i_evt)
            call stdhep_write (STDHEP_HEPEVT)
         case (FMT_STDHEP_UP)
            call event_write_to_hepeup (event)
            call stdhep_write (STDHEP_HEPEUP)
         end select
         current => current%next
      end do
    end subroutine file_list_write_event
```

Close streams.
⟨*Commands: procedures*⟩+≡
```
    subroutine file_list_close (file_list)
      type(file_list_t), intent(inout), target :: file_list
      type(file_spec_t), pointer :: current
      current => file_list%first
```

```
    do while (associated (current))
       select case (current%format)
       case (FMT_HEPMC)
          if (hepmc_is_available ()) then
             call hepmc_iostream_close (current%iostream)
             deallocate (current%iostream)
          end if
       case (FMT_LHEF)
          call les_houches_events_write_footer (current%unit)
          close (current%unit)
       case (FMT_STDHEP)
          call stdhep_end
       case (FMT_STDHEP_UP)
          call stdhep_end
       case default
          close (current%unit)
       end select
       current => current%next
    end do
  end subroutine file_list_close
```

Additional tools:

⟨*Commands: procedures*⟩+≡

```
  elemental function event_format_code (format) result (fmt)
    integer :: fmt
    type(string_t), intent(in) :: format
    select case (char (format))
    case ("ascii")
       fmt = FMT_DEFAULT
    case ("debug")
       fmt = FMT_DEBUG
    case ("hepmc")
       fmt = FMT_HEPMC
    case ("hepevt")
       fmt = FMT_HEPEVT
    case ("short")
       fmt = FMT_ASCII_SHORT
    case ("long")
       fmt = FMT_ASCII_LONG
    case ("athena")
       fmt = FMT_ATHENA
    case ("lhef")
       fmt = FMT_LHEF
    case ("lha")
       fmt = FMT_LHA
    case ("stdhep")
       fmt = FMT_STDHEP
    case ("stdhep_up")
       fmt = FMT_STDHEP_UP
    case default
       fmt = FMT_NONE
    end select
  end function event_format_code
```

16.1.5 Runtime data

These data are used and modified during the command flow. A local copy of this can be used to temporarily override defaults. The data set is transparent.

⟨*Commands: types*⟩+≡
```
    public :: rt_data_t
```

⟨*Commands: types*⟩+≡
```
    type :: rt_data_t
       type(lexer_t), pointer :: lexer => null ()
       type(var_list_t) :: var_list
       type(iterations_list_t) :: it_list
       type(iterations_list_t), dimension(:), pointer :: it_list_default
       integer, dimension(:), allocatable :: event_fmt
       type(os_data_t) :: os_data
       type(process_library_t), pointer :: prc_lib => null ()
       type(model_t), pointer :: model => null ()
       type(beam_data_t) :: beam_data
       type(lhapdf_status_t) :: lhapdf_status
       integer :: n_processes = 0
       logical :: sf_list_allocated = .false.
       type(sf_list_t), pointer  :: sf_list => null ()
       type(parse_node_t), pointer :: pn_cuts_lexpr => null ()
       type(parse_node_t), pointer :: pn_weight_expr => null ()
       type(parse_node_t), pointer :: pn_scale_expr => null ()
       type(parse_node_t), pointer :: pn_analysis_lexpr => null ()
       type(tao_random_state), pointer :: rng => null ()
       integer :: seed
       integer :: method = PRC_UNDEFINED
       logical :: quit = .false.
       integer :: quit_code = 0
    end type rt_data_t
```

Initialize runtime data. This defines special variables such as `sqrts`, and should be done only for the instance that is actually global. Local copies will inherit the special variables.

⟨*Commands: public*⟩≡
```
    public :: rt_data_global_init
```

⟨*Commands: procedures*⟩+≡
```
    subroutine rt_data_global_init (global, paths)
      type(rt_data_t), intent(out), target :: global
      type(paths_t), intent(in), optional :: paths
      logical, target, save :: known = .true.
      call os_data_init (global%os_data, paths)
      allocate (global%rng)
      call system_clock (global%seed)
      call tao_random_create (global%rng, global%seed)
      call var_list_append_int_ptr &
           (global%var_list, var_str ("seed_value"), global%seed, known, &
            intrinsic=.true.)
      call var_list_append_real &
           (global%var_list, var_str ("sqrts"), 0._default, &
            intrinsic=.true.)
```

```
call var_list_append_string &
     (global%var_list, var_str ("$model_name"), &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$restrictions"), var_str (""), &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$method"), var_str ("omega"), &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?read_color_factors"), .true., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?slha_read_input"), .true., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?slha_read_spectrum"), .true., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?slha_read_decays"), .false., &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$library_name"), &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("cm_momentum"), 0._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("cm_theta"), 0._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("cm_phi"), 0._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("luminosity"), 0._default, &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$lhapdf_file"), &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("lhapdf_member"), 0, &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("lhapdf_photon_scheme"), 0, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("isr_alpha"), 0._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("isr_q_max"), 0._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("isr_mass"), 0._default, &
      intrinsic=.true.)
```

```
call var_list_append_int &
     (global%var_list, var_str ("isr_order"), 3, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("epa_alpha"), 0._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("epa_x_min"), 0._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("epa_q_min"), 0._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("epa_e_max"), 0._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("epa_mass"), 0._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("ewa_x_min"), 0._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("ewa_q_min"), 0._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("ewa_pt_max"), 0._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("ewa_mass"), 0._default, &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?ewa_keep_momentum"), .false., &
      intrinsic=.false.)
call var_list_append_log &
     (global%var_list, var_str ("?ewa_keep_energy"), .false., &
      intrinsic=.false.)
call var_list_append_log &
     (global%var_list, var_str ("?alpha_s_is_fixed"), .true., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?alpha_s_from_lhapdf"), .false., &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("alpha_s_order"), 0, &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("alpha_s_nf"), 5, &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?alpha_s_from_mz"), .true., &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("lambda_qcd"), 200.e-3_default, &
      intrinsic=.true.)
```

```
call var_list_append_log &
     (global%var_list, var_str ("?fatal_beam_decay"), .true., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?helicity_selection_active"), .true., &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("helicity_selection_threshold"), &
      1E10_default, &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("helicity_selection_cutoff"), 1000, &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("threshold_calls"), 0, &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("min_calls_per_channel"), 10, &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("min_calls_per_bin"), 10, &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("min_bins"), 3, &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("max_bins"), 20, &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?stratified"), .true., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?use_vamp_equivalences"), .true., &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("channel_weights_power"), 0.25_default, &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$phs_file"), &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?phs_only"), .false., &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("phs_threshold_s"), 50._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("phs_threshold_t"), 100._default, &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("phs_off_shell"), 1, &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("phs_t_channel"), 2, &
```

```
          intrinsic=.true.)
    call var_list_append_real &
         (global%var_list, var_str ("phs_e_scale"), 10._default, &
          intrinsic=.true.)
    call var_list_append_real &
         (global%var_list, var_str ("phs_m_scale"), 10._default, &
          intrinsic=.true.)
    call var_list_append_real &
         (global%var_list, var_str ("phs_q_scale"), 10._default, &
          intrinsic=.true.)
    call var_list_append_log &
         (global%var_list, var_str ("?adapt_final_grids"), .true., &
          intrinsic=.true.)
    call var_list_append_log &
         (global%var_list, var_str ("?adapt_final_weights"), .false., &
          intrinsic=.true.)
    call var_list_append_log &
         (global%var_list, var_str ("?isotropic_decay"), .false., &
          intrinsic=.true.)
    call var_list_append_log &
         (global%var_list, var_str ("?diagonal_decay"), .false., &
          intrinsic=.true.)
    call var_list_append_int &
         (global%var_list, var_str ("n_events"), 0, &
          intrinsic=.true.)
    call var_list_append_log &
         (global%var_list, var_str ("?unweighted"), .true., &
          intrinsic=.true.)
    call var_list_append_string &
         (global%var_list, var_str ("$event_normalization"), var_str ("auto"),&
          intrinsic=.true.)
    call var_list_append_log &
         (global%var_list, var_str ("?negative_weights"), .false., &
          intrinsic=.true.)
    call var_list_append_log &
         (global%var_list, var_str ("?keep_beams"), .false., &
          intrinsic=.true.)
    call var_list_append_string &
         (global%var_list, var_str ("$sample"), var_str (""), &
          intrinsic=.true.)
    call var_list_append_log &
         (global%var_list, var_str ("?read_raw"), .true., &
          intrinsic=.true.)
    call var_list_append_log &
         (global%var_list, var_str ("?write_raw"), .true., &
          intrinsic=.true.)
    call var_list_append_string &
         (global%var_list, var_str ("$extension_raw"), var_str ("evx"), &
         intrinsic=.true.)
    call var_list_append_string &
         (global%var_list, var_str ("$extension_default"), var_str ("evt"), &
         intrinsic=.true.)
    call var_list_append_string &
         (global%var_list, var_str ("$extension_debug"), var_str ("debug"), &
```

```
          intrinsic=.true.)
     call var_list_append_string &
          (global%var_list, var_str ("$extension_hepevt"), var_str ("hepevt"), &
           intrinsic=.true.)
     call var_list_append_string &
          (global%var_list, var_str ("$extension_ascii_short"), var_str ("short.evt"), &
           intrinsic=.true.)
     call var_list_append_string &
          (global%var_list, var_str ("$extension_ascii_long"), var_str ("long.evt"), &
           intrinsic=.true.)
     call var_list_append_string &
          (global%var_list, var_str ("$extension_athena"), var_str ("athena.evt"), &
           intrinsic=.true.)
     call var_list_append_string &
          (global%var_list, var_str ("$extension_lhef"), var_str ("lhef"), &
           intrinsic=.true.)
     call var_list_append_string &
          (global%var_list, var_str ("$extension_lha"), var_str ("lha"), &
           intrinsic=.true.)
     call var_list_append_string &
          (global%var_list, var_str ("$extension_hepmc"), var_str ("hepmc"), &
           intrinsic=.true.)
     call var_list_append_string &
          (global%var_list, var_str ("$extension_stdhep"), var_str ("stdhep"), &
           intrinsic=.true.)
     call var_list_append_string &
          (global%var_list, var_str ("$extension_stdhep_up"), var_str ("up.stdhep"), &
           intrinsic=.true.)
     call var_list_append_string (global%var_list, &
          var_str ("$analysis_filename"), &
           intrinsic=.true.)
     call var_list_append_string (global%var_list, &
          var_str ("$label"), var_str (""), &
           intrinsic=.true.)
     call var_list_append_string (global%var_list, &
          var_str ("$physical_unit"), var_str (""), &
           intrinsic=.true.)
     call var_list_append_string (global%var_list, &
          var_str ("$title"), var_str (""), &
           intrinsic=.true.)
     call var_list_append_string (global%var_list, &
          var_str ("$description"), var_str (""), &
           intrinsic=.true.)
     call var_list_append_string (global%var_list, &
          var_str ("$xlabel"), var_str (""), &
           intrinsic=.true.)
     call var_list_append_string (global%var_list, &
          var_str ("$ylabel"), var_str (""), &
           intrinsic=.true.)
     call var_list_append_real (global%var_list, &
          var_str ("tolerance"), 0._default, &
           intrinsic=.true.)
     call rt_data_init_pointer_variables (global)
     call iterations_lists_init_default (global%it_list_default)
```

```
    end subroutine rt_data_global_init
```

This is done at compile time when a local copy of runtime data is needed:
Link the variable list and initialize all derived parameters. This allows for
synchronizing them with local variable changes without affecting global data.

Also re-initialize pointer variables, so they point to local copies of their
targets.

⟨*Commands: procedures*⟩+≡
```
    subroutine rt_data_local_init (local, global)
      type(rt_data_t), intent(inout), target :: local
      type(rt_data_t), intent(in), target :: global
      call var_list_link (local%var_list, global%var_list)
      if (associated (global%model)) then
         call var_list_init_copies (local%var_list, &
              model_get_var_list_ptr (global%model), &
              derived_only = .true.)
      end if
      call rt_data_init_pointer_variables (local)
    end subroutine rt_data_local_init
```

These variables point to objects which get local copies:

⟨*Commands: procedures*⟩+≡
```
    subroutine rt_data_init_pointer_variables (local)
      type(rt_data_t), intent(inout), target :: local
      logical, target, save :: known = .true.
      call var_list_append_string_ptr &
           (local%var_list, var_str ("$fc"), local%os_data%fc, known, &
            intrinsic=.true.)
      call var_list_append_string_ptr &
           (local%var_list, var_str ("$fcflags"), local%os_data%fcflags, known, &
           intrinsic=.true.)
    end subroutine rt_data_init_pointer_variables
```

This is done at execution time: Copy data, transfer pointers. `local` has in-
tent(inout) because its local variable list has already been prepared by the pre-
vious routine.

⟨*Commands: procedures*⟩+≡
```
    subroutine rt_data_link (local, global)
      type(rt_data_t), intent(inout), target :: local
      type(rt_data_t), intent(in), target :: global
      local%lexer => global%lexer
      call var_list_link (local%var_list, global%var_list)
      if (associated (global%model)) then
         call var_list_synchronize (local%var_list, &
              model_get_var_list_ptr (global%model), reset_pointers = .true.)
      end if
      local%it_list = global%it_list
      local%it_list_default => global%it_list_default
      if (allocated (global%event_fmt)) then
         allocate (local%event_fmt (size (global%event_fmt)))
         local%event_fmt = global%event_fmt
      end if
```

```
      local%os_data = global%os_data
      local%prc_lib => global%prc_lib
      local%model => global%model
      local%beam_data = global%beam_data
      local%lhapdf_status = global%lhapdf_status
      local%sf_list_allocated = .false.
      local%sf_list => global%sf_list
      local%pn_cuts_lexpr => global%pn_cuts_lexpr
      local%pn_weight_expr => global%pn_weight_expr
      local%pn_scale_expr => global%pn_scale_expr
      local%pn_analysis_lexpr => global%pn_analysis_lexpr
      local%rng => global%rng
    end subroutine rt_data_link
```

Restore the previous state of data; in particular, the variable list. This applies
only to model variables (which are copies); other variables are automatically
restored when local variables are removed.

Some command (`read_slha`) reads in model variables as local entities. In
this case, the original values of model variables should not be restored, but the
global variable list should by synchronized. However, this matters only if the
local model is identical to the global model; otherwise, restoring will apply to a
different model.

⟨*Commands: procedures*⟩+≡
```
    subroutine rt_data_restore (global, local, keep_model_vars)
      type(rt_data_t), intent(inout) :: global
      type(rt_data_t), intent(in) :: local
      logical, intent(in), optional :: keep_model_vars
      logical :: same_model, restore
      if (associated (global%model)) then
        same_model = &
             model_get_name (global%model) == model_get_name (local%model)
        if (present (keep_model_vars) .and. same_model) then
          restore = .not. keep_model_vars
        else
          if (.not. same_model)  call msg_message ("Restoring model '" // &
               char (model_get_name (global%model)) // "'")
          restore = .true.
        end if
        if (restore) then
          call var_list_restore (global%var_list)
        else
          call var_list_synchronize &
               (global%var_list, model_get_var_list_ptr (global%model))
        end if
      end if
    end subroutine rt_data_restore
```

Finalizer for the variable list and the structure-function list. This is done only
for the global RT dataset; local copies contain pointers to this and do not need
a finalizer.

⟨*Commands: public*⟩+≡
```
    public :: rt_data_global_final
```

⟨*Commands: procedures*⟩+≡
```
    subroutine rt_data_global_final (global)
      type(rt_data_t), intent(inout) :: global
      call var_list_final (global%var_list)
      if (global%sf_list_allocated) then
         call sf_list_final (global%sf_list)
         deallocate (global%sf_list)
         global%sf_list_allocated = .false.
      end if
      deallocate (global%it_list_default)
    end subroutine rt_data_global_final
```

16.1.6   The command type

The command type is a generic type that holds any command, compiled for
execution. It is part of a command list. These are the possibilities:

⟨*Commands: parameters*⟩+≡
```
    integer, parameter :: CMD_NONE = 0
    integer, parameter :: CMD_PROCESS = 1
    integer, parameter :: CMD_INTEGRATE = 2
    integer, parameter :: CMD_SIMULATE = 3

    integer, parameter :: CMD_COMPILE = 10
    integer, parameter :: CMD_LOAD = 11
    integer, parameter :: CMD_EXEC = 13

    integer, parameter :: CMD_BEAMS = 21

    integer, parameter :: CMD_MODEL = 31
    integer, parameter :: CMD_LIBRARY = 32
    integer, parameter :: CMD_CUTS = 33
    integer, parameter :: CMD_WEIGHT = 34
    integer, parameter :: CMD_SCALE = 35

    integer, parameter :: CMD_VAR = 41
    integer, parameter :: CMD_PRINTD = 43
    integer, parameter :: CMD_PRINTF = 44
    integer, parameter :: CMD_SHOW = 45
    integer, parameter :: CMD_EXPECT = 46
    integer, parameter :: CMD_ECHO = 47

    integer, parameter :: CMD_UNSTABLE = 51
    integer, parameter :: CMD_STABLE = 52

    integer, parameter :: CMD_SEED = 63
    integer, parameter :: CMD_ITERATIONS = 64
    integer, parameter :: CMD_SAMPLE_FORMAT = 65

    integer, parameter :: CMD_ANALYSIS = 70
    integer, parameter :: CMD_WRITE_ANALYSIS = 71
    integer, parameter :: CMD_OBSERVABLE = 72
    integer, parameter :: CMD_HISTOGRAM = 73
```

```
        integer, parameter :: CMD_PLOT = 74
        integer, parameter :: CMD_CLEAR = 75
        integer, parameter :: CMD_RECORD = 76

        integer, parameter :: CMD_INCLUDE = 91
        integer, parameter :: CMD_SCAN = 92
        integer, parameter :: CMD_IF = 93
        integer, parameter :: CMD_QUIT = 99

        integer, parameter :: CMD_SLHA = 101
```

⟨*Commands: types*⟩+≡

```
        type :: command_t
           private
           integer :: type = CMD_NONE
           type(cmd_model_t), pointer :: model => null ()
           type(cmd_library_t), pointer :: library => null ()
           type(cmd_process_t), pointer :: process => null ()
           type(cmd_compile_t), pointer :: compile => null ()
           type(cmd_load_t), pointer :: load => null ()
           type(cmd_exec_t), pointer :: exec => null ()
           type(cmd_var_t), pointer :: var => null ()
           type(cmd_slha_t), pointer :: slha => null ()
           type(cmd_printd_t), pointer :: printd => null ()
           type(cmd_printf_t), pointer :: printf => null ()
           type(cmd_show_t), pointer :: show => null ()
           type(cmd_expect_t), pointer :: expect => null ()
           type(cmd_echo_t), pointer :: echo => null ()
           type(cmd_beams_t), pointer :: beams => null ()
           type(cmd_cuts_t), pointer :: cuts => null ()
           type(cmd_weight_t), pointer :: weight => null ()
           type(cmd_scale_t), pointer :: scale => null ()
           type(cmd_seed_t), pointer :: seed => null ()
           type(cmd_iterations_t), pointer :: iterations => null ()
           type(cmd_integrate_t), pointer :: integrate => null ()
           type(cmd_observable_t), pointer :: observable => null ()
           type(cmd_histogram_t), pointer :: histogram => null ()
           type(cmd_plot_t), pointer :: plot => null ()
           type(cmd_clear_t), pointer :: clear => null ()
           type(cmd_record_t), pointer :: record => null ()
           type(cmd_analysis_t), pointer :: analysis => null ()
           type(cmd_unstable_t), pointer :: unstable => null ()
           type(cmd_stable_t), pointer :: stable => null ()
           type(cmd_sample_format_t), pointer :: events => null ()
           type(cmd_simulate_t), pointer :: simulate => null ()
           type(cmd_write_analysis_t), pointer :: write_analysis => null ()
           type(cmd_scan_t), pointer :: loop => null ()
           type(cmd_if_t), pointer :: cond => null ()
           type(cmd_include_t), pointer :: include => null ()
           type(cmd_quit_t), pointer :: quit => null ()
           type(command_t), pointer :: next => null ()
        end type command_t
```

Finalizer: Delete the specific command entry.

⟨*Commands: procedures*⟩+≡

```
    recursive subroutine command_final (command)
      type(command_t), intent(inout) :: command
      select case (command%type)
      case (CMD_NONE)
      case (CMD_MODEL)
         deallocate (command%model)
      case (CMD_LIBRARY)
         deallocate (command%library)
      case (CMD_PROCESS)
         call cmd_process_final (command%process)
         deallocate (command%process)
      case (CMD_COMPILE)
         call cmd_compile_final (command%compile)
         deallocate (command%compile)
      case (CMD_LOAD)
         call cmd_load_final (command%load)
         deallocate (command%load)
      case (CMD_EXEC)
         call cmd_exec_final (command%exec)
         deallocate (command%exec)
      case (CMD_VAR)
         call cmd_var_final (command%var)
         deallocate (command%var)
      case (CMD_SLHA)
         call cmd_slha_final (command%slha)
         deallocate (command%slha)
      case (CMD_PRINTD)
         call cmd_printd_final (command%printd)
         deallocate (command%printd)
      case (CMD_PRINTF)
         call cmd_printf_final (command%printf)
         deallocate (command%printf)
      case (CMD_SHOW)
         call cmd_show_final (command%show)
         deallocate (command%show)
      case (CMD_EXPECT)
         call cmd_expect_final (command%expect)
         deallocate (command%expect)
      case (CMD_ECHO)
         call cmd_echo_final (command%echo)
         deallocate (command%echo)
      case (CMD_BEAMS)
         call cmd_beams_final (command%beams)
         deallocate (command%beams)
      case (CMD_CUTS)
         deallocate (command%cuts)
      case (CMD_WEIGHT)
         deallocate (command%weight)
      case (CMD_SCALE)
         deallocate (command%scale)
      case (CMD_SEED)
         call cmd_seed_final (command%seed)
```

```
         deallocate (command%seed)
      case (CMD_ITERATIONS)
         call cmd_iterations_final (command%iterations)
         deallocate (command%iterations)
      case (CMD_INTEGRATE)
         call cmd_integrate_final (command%integrate)
         deallocate (command%integrate)
      case (CMD_OBSERVABLE)
         call cmd_observable_final (command%observable)
         deallocate (command%observable)
      case (CMD_HISTOGRAM)
         call cmd_histogram_final (command%histogram)
         deallocate (command%histogram)
      case (CMD_PLOT)
         call cmd_plot_final (command%plot)
         deallocate (command%plot)
      case (CMD_CLEAR)
         call cmd_clear_final (command%clear)
         deallocate (command%clear)
      case (CMD_RECORD)
         call cmd_record_final (command%record)
         deallocate (command%record)
      case (CMD_ANALYSIS)
         deallocate (command%analysis)
      case (CMD_UNSTABLE)
         call cmd_unstable_final (command%unstable)
         deallocate (command%unstable)
      case (CMD_STABLE)
         call cmd_stable_final (command%stable)
         deallocate (command%stable)
      case (CMD_SAMPLE_FORMAT)
         call cmd_sample_format_final (command%events)
         deallocate (command%events)
      case (CMD_SIMULATE)
         call cmd_simulate_final (command%simulate)
         deallocate (command%simulate)
      case (CMD_WRITE_ANALYSIS)
         call cmd_write_analysis_final (command%write_analysis)
         deallocate (command%write_analysis)
      case (CMD_SCAN)
         call cmd_scan_final (command%loop)
         deallocate (command%loop)
      case (CMD_IF)
         call cmd_if_final (command%cond)
         deallocate (command%cond)
      case (CMD_INCLUDE)
         call cmd_include_final (command%include)
         deallocate (command%include)
      case (CMD_QUIT)
         call cmd_quit_final (command%quit)
         deallocate (command%quit)
      end select
    end subroutine command_final
```

Output.

⟨*Commands: procedures*⟩+≡

```
recursive subroutine command_write (command, unit, indent)
  type(command_t), intent(in) :: command
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  select case (command%type)
  case (CMD_NONE)
     write (u, *) "[Empty command]"
  case (CMD_MODEL)
     call cmd_model_write (command%model, unit, indent)
  case (CMD_LIBRARY)
     call cmd_library_write (command%library, unit, indent)
  case (CMD_VAR)
     call cmd_var_write (command%var, unit, indent)
  case (CMD_SLHA)
     call cmd_slha_write (command%slha, unit, indent)
  case (CMD_PROCESS)
     call cmd_process_write (command%process, unit, indent)
  case (CMD_COMPILE)
     call cmd_compile_write (command%compile, unit, indent)
  case (CMD_EXEC)
     call cmd_exec_write (command%exec, unit, indent)
  case (CMD_BEAMS)
     call cmd_beams_write (command%beams, unit, indent)
  case (CMD_CUTS)
     call cmd_cuts_write (command%cuts, unit, indent)
  case (CMD_WEIGHT)
     call cmd_weight_write (command%weight, unit, indent)
  case (CMD_SCALE)
     call cmd_scale_write (command%scale, unit, indent)
  case (CMD_SEED)
     call cmd_seed_write (command%seed, unit, indent)
  case (CMD_ITERATIONS)
     call cmd_iterations_write (command%iterations, unit, indent)
  case (CMD_INTEGRATE)
     call cmd_integrate_write (command%integrate, unit, indent)
  case (CMD_OBSERVABLE)
     call cmd_observable_write (command%observable, unit, indent)
  case (CMD_HISTOGRAM)
     call cmd_histogram_write (command%histogram, unit, indent)
  case (CMD_PLOT)
     call cmd_plot_write (command%plot, unit, indent)
  case (CMD_ANALYSIS)
     call cmd_analysis_write (command%analysis, unit, indent)
  case (CMD_UNSTABLE)
     call cmd_unstable_write (command%unstable, unit, indent)
  case (CMD_STABLE)
     call cmd_stable_write (command%stable, unit, indent)
  case (CMD_SAMPLE_FORMAT)
     call cmd_sample_format_write (command%events, unit, indent)
  case (CMD_SIMULATE)
     call cmd_simulate_write (command%simulate, unit, indent)
```

```
      case (CMD_SCAN)
         call cmd_scan_write (command%loop, unit, indent)
      case (CMD_IF)
         call cmd_if_write (command%cond, unit, indent)
      case (CMD_INCLUDE)
         call cmd_include_write (command%include, unit, indent)
      case (CMD_QUIT)
         call cmd_quit_write (command%quit, unit, indent)
      end select
    end subroutine command_write
```

Compile a command. This allocates the command pointer. Some need a variable list to link to as an extra argument. (The variable list is assigned by the `model` command.)

⟨*Commands: procedures*⟩+≡
```
    recursive subroutine command_compile (command, pn, global)
      type(command_t), pointer :: command
      type(parse_node_t), intent(in), target :: pn
      type(rt_data_t), intent(inout), target :: global
      ! call parse_node_write (pn)
      allocate (command)
      select case (char (parse_node_get_rule_key (pn)))
      case ("cmd_model")
         command%type = CMD_MODEL
         call cmd_model_compile (command%model, pn, global)
      case ("cmd_library")
         command%type = CMD_LIBRARY
         call cmd_library_compile (command%library, pn)
      case ("cmd_process")
         command%type = CMD_PROCESS
         call cmd_process_compile (command%process, pn, global)
      case ("cmd_compile")
         command%type = CMD_COMPILE
         call cmd_compile_compile (command%compile, pn, global)
      case ("cmd_load")
         command%type = CMD_LOAD
         call cmd_load_compile (command%load, pn, global)
      case ("cmd_exec")
         command%type = CMD_EXEC
         call cmd_exec_compile (command%exec, pn, global)
      case ("cmd_num", "cmd_complex", "cmd_real", "cmd_int", &
            "cmd_log_decl", "cmd_log", "cmd_string", "cmd_string_decl", &
            "cmd_alias")
         command%type = CMD_VAR
         call cmd_var_compile (command%var, pn, global)
      case ("cmd_slha")
         command%type = CMD_SLHA
         call cmd_slha_compile (command%slha, pn, global)
      case ("cmd_print")
         command%type = CMD_PRINTD
         call cmd_printd_compile (command%printd, pn, global)
      case ("cmd_printf")
         command%type = CMD_PRINTF
```

```
      call cmd_printf_compile (command%printf, pn, global)
case ("cmd_show")
   command%type = CMD_SHOW
   call cmd_show_compile (command%show, pn, global)
case ("cmd_expect")
   command%type = CMD_EXPECT
   call cmd_expect_compile (command%expect, pn, global)
case ("cmd_echo")
   command%type = CMD_ECHO
   call cmd_echo_compile (command%echo, pn, global)
case ("cmd_beams")
   command%type = CMD_BEAMS
   call cmd_beams_compile (command%beams, pn, global)
case ("cmd_cuts")
   command%type = CMD_CUTS
   call cmd_cuts_compile (command%cuts, pn)
case ("cmd_weight")
   command%type = CMD_WEIGHT
   call cmd_weight_compile (command%weight, pn)
case ("cmd_scale")
   command%type = CMD_SCALE
   call cmd_scale_compile (command%scale, pn)
case ("cmd_seed")
   command%type = CMD_SEED
   call cmd_seed_compile (command%seed, pn, global)
case ("cmd_iterations")
   command%type = CMD_ITERATIONS
   call cmd_iterations_compile (command%iterations, pn, global)
case ("cmd_integrate")
   command%type = CMD_INTEGRATE
   call cmd_integrate_compile (command%integrate, pn, global)
case ("cmd_observable")
   command%type = CMD_OBSERVABLE
   call cmd_observable_compile (command%observable, pn, global)
case ("cmd_histogram")
   command%type = CMD_HISTOGRAM
   call cmd_histogram_compile (command%histogram, pn, global)
case ("cmd_plot")
   command%type = CMD_PLOT
   call cmd_plot_compile (command%plot, pn, global)
case ("cmd_clear")
   command%type = CMD_CLEAR
   call cmd_clear_compile (command%clear, pn, global)
case ("cmd_record")
   command%type = CMD_RECORD
   call cmd_record_compile (command%record, pn, global)
case ("cmd_analysis")
   command%type = CMD_ANALYSIS
   call cmd_analysis_compile (command%analysis, pn)
case ("cmd_unstable")
   command%type = CMD_UNSTABLE
   call cmd_unstable_compile (command%unstable, pn, global)
case ("cmd_stable")
   command%type = CMD_STABLE
```

```
      call cmd_stable_compile (command%stable, pn, global)
   case ("cmd_sample_format")
      command%type = CMD_SAMPLE_FORMAT
      call cmd_sample_format_compile (command%events, pn)
   case ("cmd_simulate")
      command%type = CMD_SIMULATE
      call cmd_simulate_compile (command%simulate, pn, global)
   case ("cmd_write_analysis")
      command%type = CMD_WRITE_ANALYSIS
      call cmd_write_analysis_compile (command%write_analysis, pn, global)
   case ("cmd_scan")
      command%type = CMD_SCAN
      call cmd_scan_compile (command%loop, pn, global)
   case ("cmd_if")
      command%type = CMD_IF
      call cmd_if_compile (command%cond, pn, global)
   case ("cmd_include")
      command%type = CMD_INCLUDE
      call cmd_include_compile (command%include, pn, global)
   case ("cmd_quit")
      command%type = CMD_QUIT
      call cmd_quit_compile (command%quit, pn, global)
   case default
      print *, char (parse_node_get_rule_key (pn))
      call msg_bug ("Command not implemented")
   end select
   ! call command_write (command)
   end subroutine command_compile
```

Execute a command. This will use and/or modify the runtime data set. If the
quit flag is set, the caller should terminate command execution.
⟨*Commands: procedures*⟩+≡
```
   recursive subroutine command_execute (command, global, print)
     type(command_t), intent(inout) :: command
     type(rt_data_t), intent(inout), target :: global
     logical, intent(in), optional :: print
     select case (command%type)
     case (CMD_MODEL)
        call cmd_model_execute (command%model, global)
     case (CMD_LIBRARY)
        call cmd_library_execute (command%library, global)
     case (CMD_PROCESS)
        call cmd_process_execute (command%process, global)
     case (CMD_COMPILE)
        call cmd_compile_execute (command%compile, global)
     case (CMD_LOAD)
        call cmd_load_execute (command%load, global)
     case (CMD_EXEC)
        call cmd_exec_execute (command%exec, global)
     case (CMD_VAR)
        call cmd_var_execute (command%var, global)
     case (CMD_SLHA)
        call cmd_slha_execute (command%slha, global)
```

```
case (CMD_PRINTD)
   call cmd_printd_execute (command%printd, global)
case (CMD_PRINTF)
   call cmd_printf_execute (command%printf, global)
case (CMD_SHOW)
   call cmd_show_execute (command%show, global)
case (CMD_EXPECT)
   call cmd_expect_execute (command%expect, global)
case (CMD_ECHO)
   call cmd_echo_execute (command%echo, global)
case (CMD_BEAMS)
   call cmd_beams_execute (command%beams, global)
case (CMD_CUTS)
   call cmd_cuts_execute (command%cuts, global)
case (CMD_WEIGHT)
   call cmd_weight_execute (command%weight, global)
case (CMD_SCALE)
   call cmd_scale_execute (command%scale, global)
case (CMD_SEED)
   call cmd_seed_execute (command%seed, global)
case (CMD_ITERATIONS)
   call cmd_iterations_execute (command%iterations, global)
case (CMD_INTEGRATE)
   call cmd_integrate_execute (command%integrate, global)
case (CMD_OBSERVABLE)
   call cmd_observable_execute (command%observable, global)
case (CMD_HISTOGRAM)
   call cmd_histogram_execute (command%histogram, global)
case (CMD_PLOT)
   call cmd_plot_execute (command%plot, global)
case (CMD_CLEAR)
   call cmd_clear_execute (command%clear, global)
case (CMD_RECORD)
   call cmd_record_execute (command%record, global)
case (CMD_ANALYSIS)
   call cmd_analysis_execute (command%analysis, global)
case (CMD_UNSTABLE)
   call cmd_unstable_execute (command%unstable, global)
case (CMD_STABLE)
   call cmd_stable_execute (command%stable, global)
case (CMD_SAMPLE_FORMAT)
   call cmd_sample_format_execute (command%events, global)
case (CMD_SIMULATE)
   call cmd_simulate_execute (command%simulate, global)
case (CMD_WRITE_ANALYSIS)
   call cmd_write_analysis_execute (command%write_analysis, global)
case (CMD_SCAN)
   call cmd_scan_execute (command%loop, global)
case (CMD_IF)
   call cmd_if_execute (command%cond, global)
case (CMD_INCLUDE)
   call cmd_include_execute (command%include, global)
case (CMD_QUIT)
   call cmd_quit_execute (command%quit, global)
```

```
      end select
      if (present (print)) then
         if (print)  call command_write (command)
      end if
   end subroutine command_execute
```

Auxiliary for output:

⟨*Commands: procedures*⟩+≡
```
   subroutine write_indent (unit, indent)
     integer, intent(in) :: unit
     integer, intent(in), optional :: indent
     if (present (indent)) then
        write (unit, "(A)", advance="no")  repeat ("  ", indent)
     end if
   end subroutine write_indent
```

16.1.7   Specific command types

Model configuration

The command declares a model, looks for the specified file and loads it.

⟨*Commands: types*⟩+≡
```
   type :: cmd_model_t
      private
      type(string_t) :: name
   end type cmd_model_t
```

Output

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_model_write (model, unit, indent)
     type(cmd_model_t), intent(in) :: model
     integer, intent(in), optional :: unit, indent
     integer :: u
     u = output_unit (unit);  if (u < 0)  return
     call write_indent (u, indent)
     write (u, "(1x,A,1x,'""',A,'""')")  "model =", char (model%name)
   end subroutine cmd_model_write
```

Compile. Get the model name and read the model from file, so it is readily
available when the command list is executed.

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_model_compile (model, pn, global)
     type(cmd_model_t), pointer :: model
     type(parse_node_t), intent(in), target :: pn
     type(rt_data_t), intent(inout), target :: global
     type(parse_node_t), pointer :: pn_name
     type(model_t), pointer :: mdl
     type(string_t) :: filename
     pn_name => parse_node_get_sub_ptr (pn, 3)
     allocate (model)
     if (associated (pn_name)) then
```

```
            model%name = parse_node_get_string (pn_name)
            filename = model%name // ".mdl"
            mdl => null ()
            call model_list_read_model (model%name, filename, global%os_data, mdl)
            if (associated (mdl)) then
               call var_list_init_copies &
                    (global%var_list, model_get_var_list_ptr (mdl))
            end if
         else
            model%name = ""
         end if
      end subroutine cmd_model_compile
```

Execute: Insert a pointer into the global data record and reassign the variable list.

⟨*Commands: procedures*⟩+≡
```
      subroutine cmd_model_execute (model, global)
        type(cmd_model_t), intent(in) :: model
        type(rt_data_t), intent(inout), target :: global
        type(model_t), pointer :: mdl
        type(var_list_t), pointer :: model_vars
        if (model_get_name (global%model) /= model%name) then
           if (model_list_model_exists (model%name)) then
              mdl => model_list_get_model_ptr (model%name)
              global%model => mdl
              call var_list_set_string (global%var_list, var_str ("$model_name"), &
                   model%name, is_known=.true.)
              call msg_message ("Switching to model '" &
                   // char (model_get_name (mdl)) &
                   // "': reassigning model parameters")
              model_vars => model_get_var_list_ptr (mdl)
              call var_list_synchronize &
                   (global%var_list, model_vars, reset_pointers = .true.)
           end if
        else
           model_vars => model_get_var_list_ptr (global%model)
           call var_list_synchronize &
                (global%var_list, model_vars, reset_pointers = .false.)
        end if
      end subroutine cmd_model_execute
```

Library configuration

We configure a process library that should hold the subsequently defined processes.

⟨*Commands: types*⟩+≡
```
      type :: cmd_library_t
        private
        type(string_t) :: name
      end type cmd_library_t
```

Output.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_library_write (library, unit, indent)
  type(cmd_library_t), intent(in) :: library
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,1x,'""',A,'""')")  "library =", char (library%name)
end subroutine cmd_library_write
```

Compile. Get the library name.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_library_compile (library, pn)
  type(cmd_library_t), pointer :: library
  type(parse_node_t), intent(in), target :: pn
  type(parse_node_t), pointer :: pn_name
  pn_name => parse_node_get_sub_ptr (pn, 3)
  allocate (library)
  library%name = parse_node_get_string (pn_name)
end subroutine cmd_library_compile
```

Execute: Initialize a new library and append to the library store (if it does not yet exist). Try to load the library. Insert a pointer to the library into the global data record.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_library_execute (library, global)
  type(cmd_library_t), intent(in) :: library
  type(rt_data_t), intent(inout), target :: global
  logical :: rebuild_library, recompile_library
  call process_library_store_append &
       (library%name, global%os_data, global%prc_lib)
  rebuild_library = var_list_get_lval (global%var_list, var_str ("?rebuild_library"))
  recompile_library = var_list_get_lval (global%var_list, var_str ("?recompile_library"))
  if (.not. (rebuild_library .or. recompile_library)) then
     call process_library_load (global%prc_lib, &
          global%os_data, var_list=global%var_list, ignore=.true.)
  else
     call var_list_set_string (global%var_list, var_str ("$library_name"), &
          process_library_get_name (global%prc_lib), is_known=.true.)
  end if
end subroutine cmd_library_execute
```

Process configuration

We define a process-configuration command as a specific type. The incoming and outgoing particles are given evaluation-trees which we transform to PDG-code arrays. For transferring to O'MEGA, they are reconverted to strings.

This also includes the choice of method for the corresponding process: omega for O'MEGA matrix elements, test for special processes generated by WHIZARD,

`external` for using an external matrix element and `dipole` for generating dipole subtraction terms.

⟨*Commands: types*⟩+≡

```
type :: cmd_process_t
   private
   type(string_t) :: id
   integer :: n_in  = 0
   integer :: n_out = 0
   type(eval_tree_t), dimension(:), allocatable :: pdg_in
   type(eval_tree_t), dimension(:), allocatable :: pdg_out
   type(string_t), dimension(:), allocatable :: prt_in
   type(string_t), dimension(:), allocatable :: prt_out
   type(command_list_t), pointer :: options => null ()
   type(rt_data_t) :: local
end type cmd_process_t
```

Finalize. The enclosed eval trees have to be deleted.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_process_final (process)
  type(cmd_process_t), intent(inout) :: process
  integer :: i
  if (allocated (process%pdg_in)) then
     do i = 1, size (process%pdg_in)
        call eval_tree_final (process%pdg_in(i))
     end do
  end if
  if (allocated (process%pdg_out)) then
     do i = 1, size (process%pdg_out)
        call eval_tree_final (process%pdg_out(i))
     end do
  end if
  if (associated (process%options)) then
     call command_list_final (process%options)
     deallocate (process%options)
  end if
end subroutine cmd_process_final
```

Output.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_process_write (process, unit, indent)
  type(cmd_process_t), intent(in) :: process
  integer, intent(in), optional :: unit, indent
  integer :: u, i
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,1x,A,1x,A,1x)", advance="no") &
       "process", char (process%id), "="
  do i = 1, process%n_in
     if (i /= 1)  write (u, "(',')", advance="no")
     write (u, "(1x,A)", advance="no") char (process%prt_in(i))
  end do
  write (u, "(1x,A,1x)", advance="no")  "=>"
  do i = 1, process%n_out
```

```
         if (i /= 1)  write (u, "(',')", advance="no")
         write (u, "(1x,A)", advance="no") char (process%prt_out(i))
      end do
      if (associated (process%options)) then
         write (u, "(1x,'{')")
         call command_list_write (process%options, unit, indent)
         call write_indent (u, indent)
         write (u, "(1x,'}')")
      else
         write (u, *)
      end if
   end subroutine cmd_process_write
```

Compile.

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_process_compile (process, pn, global)
     type(cmd_process_t), pointer :: process
     type(parse_node_t), intent(in), target :: pn
     type(rt_data_t), intent(in), target :: global
     type(parse_node_t), pointer :: pn_id, pn_in, pn_out, pn_codes, pn_opt
     integer :: i
     pn_id => parse_node_get_sub_ptr (pn, 2)
     pn_in  => parse_node_get_next_ptr (pn_id, 2)
     pn_out => parse_node_get_next_ptr (pn_in, 2)
     pn_opt => parse_node_get_next_ptr (pn_out)
     allocate (process)
     call rt_data_local_init (process%local, global)
     if (associated (pn_opt)) then
        allocate (process%options)
        call command_list_compile (process%options, pn_opt, process%local)
     end if
     process%id = parse_node_get_string (pn_id)
     call process_library_check_name_consistency (process%id, global%prc_lib)
     process%n_in  = parse_node_get_n_sub (pn_in)
     process%n_out = parse_node_get_n_sub (pn_out)
     pn_codes => parse_node_get_sub_ptr (pn_in)
     allocate (process%pdg_in (process%n_in), process%prt_in (process%n_in))
     do i = 1, process%n_in
        call eval_tree_init_cexpr &
             (process%pdg_in(i), pn_codes, process%local%var_list)
        process%prt_in(i) = "?"
        pn_codes => parse_node_get_next_ptr (pn_codes)
     end do
     pn_codes => parse_node_get_sub_ptr (pn_out)
     allocate (process%pdg_out (process%n_out), process%prt_out (process%n_out))
     do i = 1, process%n_out
        call eval_tree_init_cexpr &
             (process%pdg_out(i), pn_codes, process%local%var_list)
        process%prt_out(i) = "?"
        pn_codes => parse_node_get_next_ptr (pn_codes)
     end do
   end subroutine cmd_process_compile
```

Command execution. Evaluate the particle lists, transform PDG codes into strings, and add the current process configuration to the process library. If anything goes wrong in the particle code interpretation, the particle names will contain question marks. In that case, the process is not recorded.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_process_execute (process, global)
  type(cmd_process_t), intent(inout), target :: process
  type(rt_data_t), intent(inout), target :: global
  type(pdg_array_t) :: pdg_in, pdg_out
  integer :: i, method
  logical :: rebuild_library
  type(string_t) :: restrictions, method_str
  call rt_data_link (process%local, global)
  if (associated (process%options)) then
     call command_list_execute (process%options, process%local)
  end if
  if (process_library_is_static (global%prc_lib)) then
     call msg_error ("Process library '" &
           // char (process_library_get_name (global%prc_lib)) &
           // "' is static, no processes can be added")
     return
  end if
  do i = 1, size (process%pdg_in)
     call eval_tree_evaluate (process%pdg_in(i))
     pdg_in = eval_tree_get_pdg_array (process%pdg_in(i))
     process%prt_in(i) = make_flavor_string (pdg_in, process%local%model)
  end do
  do i = 1, size (process%pdg_out)
     call eval_tree_evaluate (process%pdg_out(i))
     pdg_out = eval_tree_get_pdg_array (process%pdg_out(i))
     process%prt_out(i) = make_flavor_string (pdg_out, process%local%model)
  end do
  restrictions = var_list_get_sval &
       (process%local%var_list, var_str ("$restrictions"))
  method_str = var_list_get_sval &
       (process%local%var_list, var_str ("$method"))
  method = method_of_string (method_str)
  if (all (scan (process%prt_in, "?") == 0) .and. &
       all (scan (process%prt_out, "?") == 0)) then
     rebuild_library = var_list_get_lval &
           (process%local%var_list, var_str ("?rebuild_library"))
     call process_library_append (global%prc_lib, &
           process%id, process%local%model, &
           process%prt_in, process%prt_out, &
           method = method, restrictions = restrictions, &
           rebuild_library = rebuild_library, message = .true.)
  else
     call msg_error ("Broken process declaration: skipped")
  end if
  call rt_data_restore (global, process%local)
end subroutine cmd_process_execute
```

This is a method of the eval tree, but cannot be coded inside the **expressions**

module since it uses the `model` and `flv` types which are not available there.

⟨*Commands: procedures*⟩+≡

```
function make_flavor_string (aval, model) result (prt)
  type(string_t) :: prt
  type(pdg_array_t), intent(in) :: aval
  type(model_t), intent(in), target :: model
  integer, dimension(:), allocatable :: pdg
  type(flavor_t), dimension(:), allocatable :: flv
  integer :: i
  pdg = aval
  allocate (flv (size (pdg)))
  call flavor_init (flv, pdg, model)
  if (size (pdg) /= 0) then
     prt = flavor_get_name (flv(1))
     do i = 2, size (flv)
        prt = prt // ":" // flavor_get_name (flv(i))
     end do
  else
     prt = "?"
  end if
end function make_flavor_string
```

Auxiliary function to convert the method string into an integer identifier.

⟨*Commands: procedures*⟩+≡

```
function method_of_string (meth_str) result (meth_id)
  type(string_t), intent(in) :: meth_str
  integer :: meth_id
  select case (char (meth_str))
  case ("omega")
     meth_id = PRC_OMEGA
  case ("test")
     meth_id = PRC_TEST
  case ("unit")
     meth_id = PRC_UNIT
  case ("external")
     meth_id = PRC_EXTERNAL
  case ("dipole")
     meth_id = PRC_DIPOLE
  case default
     call msg_fatal ("Invalid method for matrix elements.")
  end select
end function method_of_string
```

Process compilation

⟨*Commands: types*⟩+≡

```
type :: cmd_compile_t
  private
  type(string_t), dimension(:), allocatable :: libname
  logical :: make_executable = .false.
  type(string_t) :: exec_name
  type(command_list_t), pointer :: options => null ()
```

```
      type(rt_data_t) :: local
    end type cmd_compile_t
```

Finalize. Only options.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_compile_final (compile)
      type(cmd_compile_t), intent(inout) :: compile
      if (associated (compile%options)) then
         call command_list_final (compile%options)
         deallocate (compile%options)
      end if
    end subroutine cmd_compile_final
```

Output

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_compile_write (compile, unit, indent)
      type(cmd_compile_t), intent(in) :: compile
      integer, intent(in), optional :: unit, indent
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)", advance="no") "compile "
      if (compile%make_executable) then
         write (u, "(A)", advance="no") "as " // char (compile%exec_name) // " "
      end if
      write (u, "(A)", advance="no") "("
      do i = 1, size (compile%libname)
         if (i /= 1)  write (u, "(',',1x)", advance="no")
         write (u, "(A)", advance="no") '"' // char (compile%libname(i)) // '"'
      end do
      write (u, "(A)", advance="no") ")"
      if (associated (compile%options)) then
         write (u, "(1x,'{')")
         call command_list_write (compile%options, unit, indent)
         call write_indent (u, indent)
         write (u, "(1x,'}')")
      end if
    end subroutine cmd_compile_write
```

Compile the libraries specified in the argument. If the argument is empty,
compile all libraries which can be found in the process store.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_compile_compile (compile, pn, global)
      type(cmd_compile_t), pointer :: compile
      type(parse_node_t), intent(in), target :: pn
      type(rt_data_t), intent(in), target :: global
      type(parse_node_t), pointer :: pn_cmd, pn_clause, pn_arg, pn_lib, pn_opt
      type(parse_node_t), pointer :: pn_exec_name_spec, pn_exec_name
      type(process_library_t), pointer :: prc_lib
      integer :: n_lib, i
      pn_cmd => parse_node_get_sub_ptr (pn)
      pn_clause => parse_node_get_sub_ptr (pn_cmd)
```

```
      pn_exec_name_spec => parse_node_get_sub_ptr (pn_clause, 2)
      if (associated (pn_exec_name_spec)) then
         pn_exec_name => parse_node_get_sub_ptr (pn_exec_name_spec, 2)
      else
         pn_exec_name => null ()
      end if
      pn_arg => parse_node_get_next_ptr (pn_clause)
      pn_opt => parse_node_get_next_ptr (pn_cmd)
      allocate (compile)
      call rt_data_local_init (compile%local, global)
      if (associated (pn_opt)) then
         allocate (compile%options)
         call command_list_compile (compile%options, pn_opt, compile%local)
      end if
      if (associated (pn_arg)) then
         n_lib = parse_node_get_n_sub (pn_arg)
      else
         n_lib = 0
      end if
      if (n_lib > 0) then
         allocate (compile%libname (n_lib))
         pn_lib => parse_node_get_sub_ptr (pn_arg)
         do i = 1, n_lib
            compile%libname(i) = parse_node_get_string (pn_lib)
            pn_lib => parse_node_get_next_ptr (pn_lib)
         end do
      else
         n_lib = 0
         prc_lib => process_library_store_get_first ()
         do while (associated (prc_lib))
            n_lib = n_lib + 1
            call process_library_advance (prc_lib)
         end do
         allocate (compile%libname (n_lib))
         i = 0
         prc_lib => process_library_store_get_first ()
         do while (associated (prc_lib))
            i = i + 1
            compile%libname(i) = process_library_get_name (prc_lib)
            call process_library_advance (prc_lib)
         end do
      end if
      if (associated (pn_exec_name)) then
         compile%make_executable = .true.
         compile%exec_name = parse_node_get_string (pn_exec_name)
      end if
    end subroutine cmd_compile_compile
```

Command execution. Generate code, write driver, compile and link. Do this
for all libraries in the list which have a nonzero number of processes defined so
far.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_compile_execute (compile, global)
```

```
type(cmd_compile_t), intent(inout), target :: compile
type(rt_data_t), intent(inout), target :: global
type(string_t) :: objlist
type(process_library_t), pointer :: prc_lib
integer :: i
logical :: recompile_library
call rt_data_link (compile%local, global)
if (associated (compile%options)) then
   call command_list_execute (compile%options, compile%local)
end if
do i = 1, size (compile%libname)
   prc_lib => process_library_store_get_ptr (compile%libname(i))
   if (associated (prc_lib)) then
     if (process_library_get_n_processes (prc_lib) > 0) then
        call process_library_generate_code (prc_lib, compile%local%os_data)
        call process_library_write_driver (prc_lib)
        recompile_library = var_list_get_lval &
             (compile%local%var_list, var_str ("?recompile_library"))
        call process_library_compile &
             (prc_lib, compile%local%os_data, recompile_library, objlist)
        call process_library_link &
             (prc_lib, compile%local%os_data, objlist)
     end if
   else
     call msg_fatal ("Process library '" // char (compile%libname(i)) // &
                 "' has not been declared.")
   end if
end do
if (compile%make_executable) then
   call write_library_manager (compile%libname)
   call compile_library_manager (compile%local%os_data)
   call link_executable &
        (compile%libname, compile%exec_name, &
          get_modellibs_flags (prc_lib, compile%local%os_data), &
          compile%local%os_data)
else
   do i = 1, size (compile%libname)
      call process_library_load (prc_lib, compile%local%os_data, &
           var_list=compile%local%var_list)
   end do
end if
call rt_data_restore (global, compile%local)
end subroutine cmd_compile_execute
```

Load library

For the most part, this is analogous to the compile command.

⟨*Commands: types*⟩+≡
```
type :: cmd_load_t
  private
  type(string_t), dimension(:), allocatable :: libname
  type(command_list_t), pointer :: options => null ()
  type(rt_data_t) :: local
```

```
    end type cmd_load_t
```

Finalize. Only options.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_load_final (load)
      type(cmd_load_t), intent(inout) :: load
      if (associated (load%options)) then
         call command_list_final (load%options)
         deallocate (load%options)
      end if
    end subroutine cmd_load_final
```

Output.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_load_write (load, unit, indent)
      type(cmd_load_t), intent(in) :: load
      integer, intent(in), optional :: unit, indent
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)", advance="no") "load ("
      do i = 1, size (load%libname)
         if (i /= 1)  write (u, "(',',1x)", advance="no")
         write (u, "(A)", advance="no") '"' // char (load%libname(i)) // '"'
      end do
      write (u, "(A)", advance="no") ")"
    end subroutine cmd_load_write
```

Compile the load command.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_load_compile (load, pn, global)
      type(cmd_load_t), pointer :: load
      type(parse_node_t), intent(in), target :: pn
      type(rt_data_t), intent(in), target :: global
      type(parse_node_t), pointer :: pn_arg, pn_lib, pn_opt
      type(process_library_t), pointer :: prc_lib
      integer :: n_lib, i
      pn_arg => parse_node_get_sub_ptr (pn, 2)
      allocate (load)
      if (associated (pn_arg)) then
         select case (char (parse_node_get_rule_key (pn_arg)))
         case ("load_arg")
            n_lib = parse_node_get_n_sub (pn_arg)
            pn_opt => parse_node_get_next_ptr (pn_arg)
         case default
            n_lib = 0
            pn_opt => pn_arg
         end select
      else
         n_lib = 0
         pn_opt => null ()
      end if
```

```
          call rt_data_local_init (load%local, global)
          if (associated (pn_opt)) then
             allocate (load%options)
             call command_list_compile (load%options, pn_opt, load%local)
          end if
          if (n_lib > 0) then
             allocate (load%libname (n_lib))
             pn_lib => parse_node_get_sub_ptr (pn_arg)
             do i = 1, n_lib
                load%libname(i) = parse_node_get_string (pn_lib)
                pn_lib => parse_node_get_next_ptr (pn_lib)
             end do
          else
             n_lib = 0
             prc_lib => process_library_store_get_first ()
             do while (associated (prc_lib))
                n_lib = n_lib + 1
                call process_library_advance (prc_lib)
             end do
             allocate (load%libname (n_lib))
             i = 0
             prc_lib => process_library_store_get_first ()
             do while (associated (prc_lib))
                i = i + 1
                load%libname(i) = process_library_get_name (prc_lib)
                call process_library_advance (prc_lib)
             end do
          end if
        end subroutine cmd_load_compile
```

Execute: Load the specified shared libraries. Insert a pointer to the last library in the list into the global record.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_load_execute (load, global)
      type(cmd_load_t), intent(inout) :: load
      type(rt_data_t), intent(inout), target :: global
      type(process_library_t), pointer :: prc_lib
      integer :: i
      call rt_data_link (load%local, global)
      if (associated (load%options)) then
         call command_list_execute (load%options, load%local)
      end if
      do i = 1, size (load%libname)
         call process_library_store_append &
              (load%libname(i), load%local%os_data, prc_lib)
         call process_library_load &
              (prc_lib, load%local%os_data, var_list=load%local%var_list)
      end do
      global%prc_lib => prc_lib
      call rt_data_restore (global, load%local)
    end subroutine cmd_load_execute
```

1192

Execute a shell command

The argument is a string expression.

⟨*Commands: types*⟩+≡

```
type :: cmd_exec_t
   private
   type(eval_tree_t) :: command
end type cmd_exec_t
```

Delete the eval tree.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_exec_final (exec)
  type(cmd_exec_t), intent(inout) :: exec
  call eval_tree_final (exec%command)
end subroutine cmd_exec_final
```

Output.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_exec_write (exec, unit, indent)
  type(cmd_exec_t), intent(in) :: exec
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)", advance="no") "exec ("
  call eval_tree_write (exec%command, unit)
  write (u, "(A)") ")"
end subroutine cmd_exec_write
```

Compile the exec command.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_exec_compile (exec, pn, global)
  type(cmd_exec_t), pointer :: exec
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(in), target :: global
  type(parse_node_t), pointer :: pn_arg, pn_command
  pn_arg => parse_node_get_sub_ptr (pn, 2)
  pn_command => parse_node_get_sub_ptr (pn_arg)
  allocate (exec)
  call eval_tree_init_sexpr (exec%command, pn_command, global%var_list)
end subroutine cmd_exec_compile
```

Execute the specified shell command.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_exec_execute (exec, global)
  type(cmd_exec_t), intent(inout) :: exec
  type(rt_data_t), intent(in) :: global
  type(string_t) :: command
  integer :: status
  call eval_tree_evaluate (exec%command)
  if (eval_tree_result_is_known (exec%command)) then
     command = eval_tree_get_string (exec%command)
```

1193

```
        if (command /= "") then
            call os_system_call (command, status, verbose=.true.)
            if (status /= 0) then
                write (msg_buffer, "(A,I0)")  "Return code = ", status
                call msg_message ()
                call msg_error ("System command returned with nonzero status code")
            end if
        end if
    end if
end subroutine cmd_exec_execute
```

Variable declaration

A variable can have various types. Hold the definition as an eval tree.

⟨*Commands: types*⟩+≡
```
type :: cmd_var_t
  private
  type(string_t) :: name
  integer :: type = V_NONE
  type(eval_tree_t) :: value
  logical, pointer :: is_known => null ()
  logical, pointer :: lval => null ()
  integer, pointer :: ival => null ()
  real(default), pointer :: rval => null ()
  complex(default), pointer :: cval => null ()
  type(string_t), pointer :: sval => null ()
  type(pdg_array_t), pointer :: aval => null ()
  logical :: is_intrinsic = .false.
  logical :: is_copy = .false.
end type cmd_var_t
```

Delete the eval tree.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_var_final (var)
  type(cmd_var_t), intent(inout) :: var
  call eval_tree_final (var%value)
end subroutine cmd_var_final
```

Output

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_var_write (var, unit, indent)
  type(cmd_var_t), intent(in) :: var
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)", advance="no")  char (var%name)
  write (u, "(1x, A)")  "="
  call eval_tree_write (var%value, unit)
end subroutine cmd_var_write
```

Compile. Append the variable to the local list, compile the definition and assign
the value (if constant) or a pointer (if variable) to the newly created variable.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_var_compile (var, pn, global)
  type(cmd_var_t), pointer :: var
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_var, pn_name, pn_expr
  type(string_t) :: var_name
  type(var_entry_t), pointer :: var_entry
  integer :: type
  logical :: new
  new = .false.
  allocate (var)
  select case (char (parse_node_get_rule_key (pn)))
  case ("cmd_log_decl");    type = V_LOG
     pn_var => parse_node_get_sub_ptr (pn, 2)
     if (.not. associated (pn_var)) then   ! handle masked syntax error
        var%type = V_NONE; return
     end if
     pn_name => parse_node_get_sub_ptr (pn_var, 2)
     new = .true.
  case ("cmd_log");          type = V_LOG
     pn_name => parse_node_get_sub_ptr (pn, 2)
  case ("cmd_int");          type = V_INT
     pn_name => parse_node_get_sub_ptr (pn, 2)
     new = .true.
  case ("cmd_real");         type = V_REAL
     pn_name => parse_node_get_sub_ptr (pn, 2)
     new = .true.
  case ("cmd_complex");      type = V_CMPLX
     pn_name => parse_node_get_sub_ptr (pn, 2)
     new = .true.
  case ("cmd_num");          type = V_NONE
     pn_name => parse_node_get_sub_ptr (pn)
  case ("cmd_string_decl"); type = V_STR
     pn_var => parse_node_get_sub_ptr (pn, 2)
     if (.not. associated (pn_var)) then   ! handle masked syntax error
        var%type = V_NONE; return
     end if
     pn_name => parse_node_get_sub_ptr (pn_var, 2)
     new = .true.
  case ("cmd_string");       type = V_STR
     pn_name => parse_node_get_sub_ptr (pn, 2)
  case ("cmd_alias");        type = V_PDG
     pn_name => parse_node_get_sub_ptr (pn, 2)
     new = .true.
  case default
     call parse_node_mismatch &
          ("logical|int|real|complex|?|$|alias|var_name", pn)  ! $
  end select
  if (.not. associated (pn_name)) then   ! handle masked syntax error
     var%type = V_NONE; return
  end if
```

```
var_name = parse_node_get_string (pn_name)
select case (type)
case (V_LOG);  var_name = "?" // var_name
case (V_STR);  var_name = "$" // var_name    ! $
end select
call var_list_check_user_var (global%var_list, var_name, type, new)
var%type = type
var%name = var_name
var_entry => var_list_get_var_ptr &
   (global%var_list, var%name, var%type, follow_link=.false.)
if (associated (var_entry)) then
   var%is_copy = var_entry_is_copy (var_entry)
else
   var_entry => var_list_get_var_ptr &
      (global%var_list, var%name, var%type, follow_link=.true.)
   if (associated (var_entry)) then
      var%is_intrinsic = var_entry_is_intrinsic (var_entry)
      if (var_entry_is_copy (var_entry)) then
         var%is_copy = .true.
         call var_list_init_copy (global%var_list, var_entry)
      end if
   end if
   if (.not. var%is_copy) then
      select case (var%type)
      case (V_LOG)
         call var_list_append_log (global%var_list, var%name, &
               intrinsic=var%is_intrinsic)
      case (V_INT)
         call var_list_append_int (global%var_list, var%name, &
               intrinsic=var%is_intrinsic)
      case (V_REAL)
         call var_list_append_real (global%var_list, var%name, &
               intrinsic=var%is_intrinsic)
      case (V_CMPLX)
         call var_list_append_cmplx (global%var_list, var%name, &
               intrinsic=var%is_intrinsic)
      case (V_PDG)
         call var_list_append_pdg_array (global%var_list, var%name, &
               intrinsic=var%is_intrinsic)
      case (V_STR)
         call var_list_append_string (global%var_list, var%name, &
               intrinsic=var%is_intrinsic)
      end select
   end if
end if
pn_expr => parse_node_get_next_ptr (pn_name, 2)
if (associated (pn_expr)) then
   select case (var%type)
   case (V_LOG)
      call eval_tree_init_lexpr (var%value, pn_expr, global%var_list)
      var%lval => eval_tree_get_log_ptr (var%value)
   case (V_INT)
      call eval_tree_init_expr (var%value, pn_expr, global%var_list)
      call eval_tree_convert_result (var%value, var%type)
```

```
                var%ival => eval_tree_get_int_ptr (var%value)
          case (V_REAL)
             call eval_tree_init_expr (var%value, pn_expr, global%var_list)
             call eval_tree_convert_result (var%value, var%type)
             var%rval => eval_tree_get_real_ptr (var%value)
          case (V_CMPLX)
             call eval_tree_init_expr (var%value, pn_expr, global%var_list)
             call eval_tree_convert_result (var%value, var%type)
             var%cval => eval_tree_get_cmplx_ptr (var%value)
          case (V_PDG)
             call eval_tree_init_cexpr (var%value, pn_expr, global%var_list)
             var%aval => eval_tree_get_pdg_array_ptr (var%value)
          case (V_STR)
             call eval_tree_init_sexpr (var%value, pn_expr, global%var_list)
             var%sval => eval_tree_get_string_ptr (var%value)
          end select
          var%is_known => eval_tree_result_is_known_ptr (var%value)
       else
          var%type = V_NONE
       end if
    end subroutine cmd_var_compile
```

Execute. Evaluate the definition and assign the variable value. If the variable
is a copy, the original is a model variable. The original is set automatically, and
an update of the dependent parameters is in order.

⟨*Commands: procedures*⟩+≡

```
    subroutine cmd_var_execute (var, global)
      type(cmd_var_t), intent(inout), target :: var
      type(rt_data_t), intent(inout), target :: global
      type(string_t) :: model_name
      type(var_list_t), pointer :: model_vars
      type(var_entry_t) :: model_var
      if (eval_tree_is_defined (var%value)) then
         call eval_tree_evaluate (var%value)
         if (associated (global%model)) then
            model_name = model_get_name (global%model)
            model_vars => model_get_var_list_ptr (global%model)
            if (var%is_copy) then
               call var_list_set_original_pointer (global%var_list, var%name, &
                    model_vars)
            end if
            select case (var%type)
            case (V_LOG)
               call var_list_set_log (global%var_list, var%name, &
                    var%lval, var%is_known, verbose=.true., model_name=model_name)
            case (V_INT)
               call var_list_set_int (global%var_list, var%name, &
                    var%ival, var%is_known, verbose=.true., model_name=model_name)
            case (V_REAL)
               call var_list_set_real (global%var_list, var%name, &
                    var%rval, var%is_known, verbose=.true., model_name=model_name)
            case (V_CMPLX)
               call var_list_set_cmplx (global%var_list, var%name, &
```

```
                    var%cval, var%is_known, verbose=.true., model_name=model_name)
            case (V_PDG)
               call var_list_set_pdg_array (global%var_list, var%name, &
                    var%aval, var%is_known, verbose=.true., model_name=model_name)
            case (V_STR)
               call var_list_set_string (global%var_list, var%name, &
                    var%sval, var%is_known, verbose=.true., model_name=model_name)
            end select
            if (var%is_copy) then
               call model_parameters_update (global%model)
               call var_list_synchronize (global%var_list, model_vars)
            end if
         else
            select case (var%type)
            case (V_LOG)
               call var_list_set_log (global%var_list, var%name, &
                    var%lval, var%is_known, verbose=.true.)
            case (V_INT)
               call var_list_set_int (global%var_list, var%name, &
                    var%ival, var%is_known, verbose=.true.)
            case (V_REAL)
               call var_list_set_real (global%var_list, var%name, &
                    var%rval, var%is_known, verbose=.true.)
            case (V_CMPLX)
               call var_list_set_cmplx (global%var_list, var%name, &
                    var%cval, var%is_known, verbose=.true.)
            case (V_PDG)
               call var_list_set_pdg_array (global%var_list, var%name, &
                    var%aval, var%is_known, verbose=.true.)
            case (V_STR)
               call var_list_set_string (global%var_list, var%name, &
                    var%sval, var%is_known, verbose=.true.)
            end select
         end if
      else
         call msg_error ("setting variable '" // char (var%name) &
              // "': right-hand side is undefined")
      end if
    end subroutine cmd_var_execute
```

SLHA

Read a SLHA (SUSY Les Houches Accord) file to fill the appropriate model
parameters. We do not access the current variable record, but directly work on
the appropriate SUSY model, which is loaded if necessary.

We may be in read or write mode. In the latter case, we may write just
input parameters, or the complete spectrum, or the spectrum with all decays.

⟨*Commands: types*⟩+≡
```
    type :: cmd_slha_t
       private
       type(string_t) :: file
       logical :: write = .false.
```

```
      type(command_list_t), pointer :: options => null ()
      type(rt_data_t) :: local
    end type cmd_slha_t
```

Finalizer.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_slha_final (slha)
      type(cmd_slha_t), intent(inout) :: slha
      if (associated (slha%options)) then
         call command_list_final (slha%options)
         deallocate (slha%options)
      end if
    end subroutine cmd_slha_final
```

Output

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_slha_write (slha, unit, indent)
      type(cmd_slha_t), intent(in) :: slha
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      if (slha%write) then
         write (u, "(A)", advance="no")  "write_"
      else
         write (u, "(A)", advance="no")  "read_"
      end if
      write (u, "(A)", advance="no")  "slha"
      write (u, "(1x,A)")  "(" // char (slha%file) // ")"
      if (associated (slha%options)) then
         write (u, "(1x,'{')")
         call command_list_write (slha%options, unit, indent)
         call write_indent (u, indent)
         write (u, "(1x,'}')")
      end if
    end subroutine cmd_slha_write
```

Compile. Read the filename and store it.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_slha_compile (slha, pn, global)
      type(cmd_slha_t), pointer :: slha
      type(parse_node_t), intent(in), target :: pn
      type(rt_data_t), intent(in), target :: global
      type(parse_node_t), pointer :: pn_key, pn_arg, pn_file
      type(parse_node_t), pointer :: pn_opt
      pn_key => parse_node_get_sub_ptr (pn)
      pn_arg => parse_node_get_next_ptr (pn_key)
      pn_file => parse_node_get_sub_ptr (pn_arg)
      pn_opt => parse_node_get_next_ptr (pn_arg)
      allocate (slha)
      call rt_data_local_init (slha%local, global)
      if (associated (pn_opt)) then
```

```
       allocate (slha%options)
       call command_list_compile (slha%options, pn_opt, slha%local)
    end if
    select case (char (parse_node_get_key (pn_key)))
    case ("read_slha")
       slha%write = .false.
    case ("write_slha")
       slha%write = .true.
    case default
       call parse_node_mismatch ("read_slha|write_slha",  pn)
    end select
    slha%file = parse_node_get_string (pn_file)
  end subroutine cmd_slha_compile
```

Execute. Read or write the specified SLHA file.

⟨*Commands: procedures*⟩+≡
```
  subroutine cmd_slha_execute (slha, global)
    type(cmd_slha_t), intent(inout), target :: slha
    type(rt_data_t), intent(inout), target :: global
    type(model_t), pointer :: mdl
    logical :: input, spectrum, decays
    call rt_data_link (slha%local, global)
    if (associated (slha%options)) then
       call command_list_execute (slha%options, slha%local)
    end if
    if (slha%write) then
       input = .true.
       spectrum = .false.
       decays = .false.
       call slha_write_file &
            (slha%file, slha%local%model, &
             input = input, spectrum = spectrum, decays = decays)
    else
       input = var_list_get_lval (slha%local%var_list, &
            var_str ("?slha_read_input"))
       spectrum = var_list_get_lval (slha%local%var_list, &
            var_str ("?slha_read_spectrum"))
       decays = var_list_get_lval (slha%local%var_list, &
            var_str ("?slha_read_decays"))
       call slha_read_file &
            (slha%file, slha%local%os_data, slha%local%model, &
             input = input, spectrum = spectrum, decays = decays)
    end if
    call rt_data_restore (global, slha%local, keep_model_vars = .true.)
  end subroutine cmd_slha_execute
```

Print default-formatted values

⟨*Commands: types*⟩+≡
```
  type :: cmd_printd_t
     private
     type(eval_tree_t) :: sexpr
```

```
      type(command_list_t), pointer :: options => null ()
      type(rt_data_t) :: local
   end type cmd_printd_t
```

Finalize.

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_printd_final (printd)
     type(cmd_printd_t), intent(inout) :: printd
     call eval_tree_final (printd%sexpr)
     if (associated (printd%options)) then
        call command_list_final (printd%options)
        deallocate (printd%options)
     end if
   end subroutine cmd_printd_final
```

Compile. We create a fake parse node (subtree) with a `sprintd` command with identical arguments which can then be handled by the corresponding evaluation procedure.

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_printd_compile (printd, pn, global)
     type(cmd_printd_t), pointer :: printd
     type(parse_node_t), intent(in), target :: pn
     type(rt_data_t), intent(in), target :: global
     type(parse_node_t), pointer :: pn_cmd, pn_arg, pn_opt
     type(parse_node_t), pointer :: pn_sexpr, pn_sprintd
     type(string_t) :: key
     integer :: i, n_args
     pn_cmd => parse_node_get_sub_ptr (pn)
     pn_opt => parse_node_get_next_ptr (pn_cmd)
     pn_arg => parse_node_get_sub_ptr (pn_cmd)
     allocate (printd)
     call rt_data_local_init (printd%local, global)
     if (associated (pn_opt)) then
        allocate (printd%options)
        call command_list_compile (printd%options, pn_opt, printd%local)
     end if
     call parse_node_create_branch (pn_sexpr, &
          syntax_get_rule_ptr (syntax_cmd_list, var_str ("sexpr")))
     call parse_node_create_branch (pn_sprintd, &
          syntax_get_rule_ptr (syntax_cmd_list, var_str ("sprintd_fun")))
     call parse_node_append_sub (pn_sexpr, pn_sprintd)
     call parse_node_append_sub (pn_sprintd, pn_arg)
     call eval_tree_init_sexpr (printd%sexpr, pn_sexpr, printd%local%var_list)
     call parse_node_final (pn_sprintd, recursive = .false.)
     call parse_node_final (pn_sexpr, recursive = .false.)
   end subroutine cmd_printd_compile
```

Execute. Evaluate the string (pretending this is a `sprintd` expression) and print it.

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_printd_execute (printd, global)
     type(cmd_printd_t), intent(inout) :: printd
```

```
      type(rt_data_t), intent(inout), target :: global
      type(string_t) :: name
      type(process_library_t), pointer :: prc_lib
      integer :: i
      call rt_data_link (printd%local, global)
      if (associated (printd%options)) then
         call command_list_execute (printd%options, printd%local)
      end if
      call eval_tree_evaluate (printd%sexpr)
      call msg_result (char (eval_tree_get_string (printd%sexpr)))
      call rt_data_restore (global, printd%local)
   end subroutine cmd_printd_execute
```

Print custom-formatted values

⟨*Commands: types*⟩+≡
```
   type :: cmd_printf_t
     private
     type(eval_tree_t) :: sexpr
     type(command_list_t), pointer :: options => null ()
     type(rt_data_t) :: local
   end type cmd_printf_t
```

Finalize.

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_printf_final (printf)
     type(cmd_printf_t), intent(inout) :: printf
     call eval_tree_final (printf%sexpr)
     if (associated (printf%options)) then
        call command_list_final (printf%options)
        deallocate (printf%options)
     end if
   end subroutine cmd_printf_final
```

Compile. We create a fake parse node (subtree) with a `sprintf` command with
identical arguments which can then be handled by the corresponding evaluation
procedure.

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_printf_compile (printf, pn, global)
     type(cmd_printf_t), pointer :: printf
     type(parse_node_t), intent(in), target :: pn
     type(rt_data_t), intent(in), target :: global
     type(parse_node_t), pointer :: pn_cmd, pn_clause, pn_arg, pn_opt
     type(parse_node_t), pointer :: pn_sexpr, pn_sprintf
     type(string_t) :: key
     integer :: i, n_args
     pn_cmd => parse_node_get_sub_ptr (pn)
     pn_opt => parse_node_get_next_ptr (pn_cmd)
     pn_clause => parse_node_get_sub_ptr (pn_cmd)
     pn_arg => parse_node_get_next_ptr (pn_clause)
     allocate (printf)
```

```
        call rt_data_local_init (printf%local, global)
        if (associated (pn_opt)) then
           allocate (printf%options)
           call command_list_compile (printf%options, pn_opt, printf%local)
        end if
        call parse_node_create_branch (pn_sexpr, &
             syntax_get_rule_ptr (syntax_cmd_list, var_str ("sexpr")))
        call parse_node_create_branch (pn_sprintf, &
             syntax_get_rule_ptr (syntax_cmd_list, var_str ("sprintf_fun")))
        call parse_node_append_sub (pn_sexpr, pn_sprintf)
        call parse_node_append_sub (pn_sprintf, pn_clause)
        call eval_tree_init_sexpr (printf%sexpr, pn_sexpr, printf%local%var_list)
        call parse_node_final (pn_sprintf, recursive = .false.)
        call parse_node_final (pn_sexpr, recursive = .false.)
      end subroutine cmd_printf_compile
```

Execute. Evaluate the string (pretending this is a `sprintf` expression) and print it.

⟨*Commands: procedures*⟩+≡

```
      subroutine cmd_printf_execute (printf, global)
        type(cmd_printf_t), intent(inout) :: printf
        type(rt_data_t), intent(inout), target :: global
        type(string_t) :: name
        type(process_library_t), pointer :: prc_lib
        integer :: i
        call rt_data_link (printf%local, global)
        if (associated (printf%options)) then
           call command_list_execute (printf%options, printf%local)
        end if
        call eval_tree_evaluate (printf%sexpr)
        call msg_result (char (eval_tree_get_string (printf%sexpr)))
        call rt_data_restore (global, printf%local)
      end subroutine cmd_printf_execute
```

Show values of variables

⟨*Commands: types*⟩+≡

```
      type :: cmd_show_t
        private
        type(string_t), dimension(:), allocatable :: name
        type(eval_tree_t), dimension(:), allocatable :: expr
        type(var_entry_t), dimension(:), allocatable :: value
      end type cmd_show_t
```

Finalize.

⟨*Commands: procedures*⟩+≡

```
      subroutine cmd_show_final (show)
        type(cmd_show_t), intent(inout) :: show
        integer :: i
        if (allocated (show%expr)) then
           do i = 1, size (show%expr)
```

```
            call eval_tree_final (show%expr(i))
         end do
      end if
      if (allocated (show%value)) then
         do i = 1, size (show%value)
            call var_entry_final (show%value(i))
         end do
      end if
   end subroutine cmd_show_final
```

Compile. Allocate an array which is filled with the names of the variables to
show.

⟨*Commands: procedures*⟩+≡

```
   subroutine cmd_show_compile (show, pn, global)
     type(cmd_show_t), pointer :: show
     type(parse_node_t), intent(in), target :: pn
     type(rt_data_t), intent(in), target :: global
     type(parse_node_t), pointer :: pn_arg, pn_var, pn_prefix, pn_name
     type(string_t) :: key
     integer :: i, n_args
     pn_arg => parse_node_get_sub_ptr (pn, 2)
     allocate (show)
     if (associated (pn_arg)) then
        n_args = parse_node_get_n_sub (pn_arg)
        allocate (show%name (n_args), show%expr (n_args), show%value (n_args))
        pn_var => parse_node_get_sub_ptr (pn_arg)
        i = 0
        do while (associated (pn_var))
           i = i + 1
           select case (char (parse_node_get_rule_key (pn_var)))
           case ("model", "beams", "results", "unstable", &
                 "real", "int", "intrinsic", &
                 "cuts", "weight", "scale", "analysis", &
                 "expect")
              show%name(i) = parse_node_get_key (pn_var)
           case ("library_spec")
              pn_prefix => parse_node_get_sub_ptr (pn_var)
              pn_name => parse_node_get_next_ptr (pn_prefix)
              key = parse_node_get_key (pn_prefix)
              if (associated (pn_name)) then
                 show%name(i) = "L " // parse_node_get_string (pn_name)
              else
                 show%name(i) = key
              end if
           case ("result_var")
              pn_prefix => parse_node_get_sub_ptr (pn_var)
              pn_name => parse_node_get_next_ptr (pn_prefix)
              if (associated (pn_name)) then
                 show%name(i) = parse_node_get_key (pn_prefix) &
                       // "(" // parse_node_get_string (pn_name) // ")"
              else
                 show%name(i) = parse_node_get_key (pn_prefix)
              end if
```

```
           case ("log_var", "alias_var", "string_var")
              pn_prefix => parse_node_get_sub_ptr (pn_var)
              pn_name => parse_node_get_next_ptr (pn_prefix)
              key = parse_node_get_key (pn_prefix)
              if (associated (pn_name)) then
                 select case (char (parse_node_get_rule_key (pn_name)))
                 case ("variable")
                    select case (char (key))
                    case ("?", "$")  ! $ sign
                       show%name(i) = key // parse_node_get_string (pn_name)
                    case ("alias")
                       show%name(i) = "A " // parse_node_get_string (pn_name)
                    case ("library")
                       show%name(i) = "L " // parse_node_get_string (pn_name)
                    end select
                 case ("lexpr")
                    show%name(i) = "<expr>"
                    call eval_tree_init_lexpr &
                         (show%expr(i), pn_name, global%var_list)
                    call var_entry_init_log_ptr (show%value(i), &
                         var_str ("logical value"), &
                         eval_tree_get_log_ptr (show%expr(i)), &
                         eval_tree_result_is_known_ptr (show%expr(i)))
                 case ("cexpr")
                    show%name(i) = "<expr>"
                    call eval_tree_init_cexpr &
                         (show%expr(i), pn_name, global%var_list)
                    call var_entry_init_pdg_array_ptr (show%value(i), &
                         var_str ("PDG-array value"), &
                         eval_tree_get_pdg_array_ptr (show%expr(i)), &
                         eval_tree_result_is_known_ptr (show%expr(i)))
                 case ("sexpr")
                    show%name(i) = "<expr>"
                    call eval_tree_init_sexpr &
                         (show%expr(i), pn_name, global%var_list)
                    call var_entry_init_string_ptr (show%value(i), &
                         var_str ("string value"), &
                         eval_tree_get_string_ptr (show%expr(i)), &
                         eval_tree_result_is_known_ptr (show%expr(i)))
                 case default
                    call parse_node_mismatch &
                         ("variable|expr|lexpr|cexpr|sexpr",  pn_name)
                 end select
              else
                 show%name(i) = key
              end if
           case ("num_var")
              pn_name => parse_node_get_sub_ptr (pn_var)
              if (associated (pn_name)) then
                 select case (char (parse_node_get_rule_key (pn_name)))
                 case ("variable")
                    show%name(i) = parse_node_get_string (pn_name)
                 case ("expr")
                    show%name(i) = "<expr>"
```

```
                         call eval_tree_init_expr &
                              (show%expr(i), pn_name, global%var_list)
                         select case (eval_tree_get_result_type (show%expr(i)))
                         case (V_INT)
                            call var_entry_init_int_ptr (show%value(i), &
                                 var_str ("integer value"), &
                                 eval_tree_get_int_ptr (show%expr(i)), &
                                 eval_tree_result_is_known_ptr (show%expr(i)))
                         case (V_REAL)
                            call var_entry_init_real_ptr (show%value(i), &
                                 var_str ("real value"), &
                                 eval_tree_get_real_ptr (show%expr(i)), &
                                 eval_tree_result_is_known_ptr (show%expr(i)))
                         case (V_CMPLX)
                            call var_entry_init_cmplx_ptr (show%value(i), &
                                 var_str ("complex value"), &
                                 eval_tree_get_cmplx_ptr (show%expr(i)), &
                                 eval_tree_result_is_known_ptr (show%expr(i)))
                         end select
                      case default
                         call parse_node_mismatch &
                              ("variable|expr",  pn_name)
                      end select
                   else
                      show%name(i) = key
                   end if
                case default
                   pn_prefix => null ()
                   pn_name => parse_node_get_sub_ptr (pn_var)
                   if (associated (pn_name)) then
                      show%name(i) = parse_node_get_string (pn_name)
                   else
                      show%name(i) = ""
                   end if
                end select
                pn_var => parse_node_get_next_ptr (pn_var)
             end do
          else
             allocate (show%name (0))
          end if
        end subroutine cmd_show_compile
```

Execute. Scan the list of variables to show. Special cases of an empty list (show all) and keywords int, real, alias have to be handled as well.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_show_execute (show, global)
      type(cmd_show_t), intent(inout) :: show
      type(rt_data_t), intent(in), target :: global
      type(string_t) :: name
      type(process_library_t), pointer :: prc_lib
      integer :: i, u
      u = logfile_unit ()
      if (size (show%name) == 0) then
```

```
         call var_list_write (global%var_list)
else
   if (associated (global%model)) then
      name = model_get_name (global%model)
   else
      name = "[undefined]"
   end if
   do i = 1, size (show%name)
      select case (char (show%name(i)))
      case ("model")
         print *, "model* = ", char (name)
         write (u, *)  "model* = ", char (name)
      case ("library")
         if (associated (global%prc_lib)) then
            call process_library_write (global%prc_lib)
            call process_library_write (global%prc_lib, unit=u)
         else
            call msg_message ("Show library: no library is loaded")
         end if
      case ("beams")
         call beam_data_write (global%beam_data)
         call beam_data_write (global%beam_data, unit=u)
      case ("results")
         call process_store_write_results ()
         call process_store_write_results (unit=u)
      case ("unstable")
         call decay_store_write ()
         call decay_store_write (unit=u)
      case ("cuts")
         if (associated (global%pn_cuts_lexpr)) then
            call parse_node_write_rec (global%pn_cuts_lexpr)
            call parse_node_write_rec (global%pn_cuts_lexpr, u)
         else
            call msg_message ("No cut expression defined")
         end if
      case ("weight")
         if (associated (global%pn_weight_expr)) then
            call parse_node_write_rec (global%pn_weight_expr)
            call parse_node_write_rec (global%pn_weight_expr,u)
         else
            call msg_message ("No weight expression defined")
         end if
      case ("scale")
         if (associated (global%pn_scale_expr)) then
            call parse_node_write_rec (global%pn_scale_expr)
            call parse_node_write_rec (global%pn_scale_expr,u)
         else
            call msg_message ("No scale expression defined")
         end if
      case ("analysis")
         if (associated (global%pn_analysis_lexpr)) then
            call parse_node_write_rec (global%pn_analysis_lexpr)
            call parse_node_write_rec (global%pn_analysis_lexpr, u)
         else
```

```
            call msg_message ("No cut expression defined")
         end if
      case ("expect")
         call expect_summary ()
      case ("?")
         if (associated (global%model)) then
            call var_list_write (global%var_list, only_type=V_LOG, &
                  model_name = name)
            call var_list_write (global%var_list, only_type=V_LOG, &
                  model_name = name, unit=u)
         else
            call var_list_write (global%var_list, only_type=V_LOG)
            call var_list_write (global%var_list, only_type=V_LOG, unit=u)
         end if
      case ("intrinsic")
         if (associated (global%model)) then
            call var_list_write (global%var_list, intrinsic=.true., &
                  model_name = name)
            call var_list_write (global%var_list, intrinsic=.true., &
                  model_name = name, unit=u)
         else
            call var_list_write (global%var_list, intrinsic=.true.)
            call var_list_write (global%var_list, intrinsic=.true., unit=u)
         end if
      case ("int")
         if (associated (global%model)) then
            call var_list_write (global%var_list, only_type=V_INT, &
                  model_name = name)
            call var_list_write (global%var_list, only_type=V_INT, &
                  model_name = name, unit=u)
         else
            call var_list_write (global%var_list, only_type=V_INT)
            call var_list_write (global%var_list, only_type=V_INT, unit=u)
         end if
      case ("real")
         if (associated (global%model)) then
            call var_list_write (global%var_list, only_type=V_REAL, &
                  model_name = name)
            call var_list_write (global%var_list, only_type=V_REAL, &
                  model_name = name, unit=u)
         else
            call var_list_write (global%var_list, only_type=V_REAL)
            call var_list_write (global%var_list, only_type=V_REAL, unit=u)
         end if
      case ("complex")
         if (associated (global%model)) then
            call var_list_write (global%var_list, only_type=V_CMPLX, &
                  model_name = name)
            call var_list_write (global%var_list, only_type=V_CMPLX, &
                  model_name = name, unit=u)
         else
            call var_list_write (global%var_list, only_type=V_CMPLX)
            call var_list_write (global%var_list, only_type=V_CMPLX, unit=u)
         end if
```

1208

```
            case ("alias")
               if (associated (global%model)) then
                  call var_list_write (global%var_list, only_type=V_PDG, &
                       model_name = name)
                  call var_list_write (global%var_list, only_type=V_PDG, &
                       model_name = name, unit=u)
               else
                  call var_list_write (global%var_list, only_type=V_PDG)
                  call var_list_write (global%var_list, only_type=V_PDG, unit=u)
               end if
            case ("$")  !$ sign
               if (associated (global%model)) then
                  call var_list_write (global%var_list, only_type=V_STR, &
                       model_name = name)
                  call var_list_write (global%var_list, only_type=V_STR, &
                       model_name = name, unit=u)
               else
                  call var_list_write (global%var_list, only_type=V_STR)
                  call var_list_write (global%var_list, only_type=V_STR, unit=u)
               end if
            case ("n_calls", &
                 "integral", "error", "accuracy", "chi2", "efficiency")
               call var_list_write (global%var_list, prefix=char(show%name(i)))
               call var_list_write (global%var_list, prefix=char(show%name(i)), &
                    unit=u)
            case ("<expr>")
               call eval_tree_evaluate (show%expr(i))
               call var_entry_write (show%value(i))
               call var_entry_write (show%value(i), unit=u)
            case default
               select case (char (extract (show%name(i), 1, 2)))
               case ("L ")
                  name = extract (show%name(i), 3)
                  prc_lib => process_library_store_get_ptr (name)
                  if (associated (prc_lib)) then
                     call process_library_write (prc_lib)
                     call process_library_write (prc_lib, unit=u)
                  else
                     call msg_message ("Library '" // char (name) &
                          // "' not loaded")
                  end if
               case ("A ")
                  name = extract (show%name(i), 3)
                  call var_list_write_var (global%var_list, name, &
                       model_name = name, type = V_PDG)
                  call var_list_write_var (global%var_list, name, &
                       model_name = name, type = V_PDG, unit=u)
               case default
                  if (associated (global%model)) then
                     call var_list_write_var (global%var_list, show%name(i), &
                          model_name = name)
                     call var_list_write_var (global%var_list, show%name(i), &
                          model_name = name, unit=u)
                  else
```

```
                      call var_list_write_var (global%var_list, show%name(i))
                      call var_list_write_var (global%var_list, show%name(i), &
                           unit=u)
                 end if
              end select
           end select
        end do
     end if
     flush (u)
  end subroutine cmd_show_execute
```

Compare values of variables to expectation

The implementation is similar to the `show` command. There are just two arguments: two values that should be compared. For providing local values for the numerical tolerance, the command has a local argument list.

   If the expectation fails, an error condition is recorded.

⟨*Commands: types*⟩+≡
```
  type :: cmd_expect_t
     private
     type(eval_tree_t) :: lexpr
     type(command_list_t), pointer :: options => null ()
     type(rt_data_t) :: local
  end type cmd_expect_t
```

Finalize.

⟨*Commands: procedures*⟩+≡
```
  subroutine cmd_expect_final (expect)
    type(cmd_expect_t), intent(inout) :: expect
    call eval_tree_final (expect%lexpr)
    if (associated (expect%options)) then
       call command_list_final (expect%options)
       deallocate (expect%options)
    end if
  end subroutine cmd_expect_final
```

Compile. Allocate an array which is filled with the names of the variables to expect.

⟨*Commands: procedures*⟩+≡
```
  subroutine cmd_expect_compile (expect, pn, global)
    type(cmd_expect_t), pointer :: expect
    type(parse_node_t), intent(in), target :: pn
    type(rt_data_t), intent(in), target :: global
    type(parse_node_t), pointer :: pn_arg, pn_expr, pn_opt
    pn_arg => parse_node_get_sub_ptr (pn, 2)
    pn_opt => parse_node_get_next_ptr (pn_arg)
    allocate (expect)
    call rt_data_local_init (expect%local, global)
    if (associated (pn_opt)) then
       allocate (expect%options)
       call command_list_compile (expect%options, pn_opt, expect%local)
```

```
      end if
      pn_expr => parse_node_get_sub_ptr (pn_arg)
      call eval_tree_init_lexpr (expect%lexpr, pn_expr, expect%local%var_list)
    end subroutine cmd_expect_compile
```

Execute. Evaluate both arguments, print them and their difference (if numerical), and whether they agree. Record the result.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_expect_execute (expect, global)
      type(cmd_expect_t), intent(inout) :: expect
      type(rt_data_t), intent(inout), target :: global
      logical :: success
      integer :: u
      u = logfile_unit ()
      call rt_data_link (expect%local, global)
      if (associated (expect%options)) then
        call command_list_execute (expect%options, expect%local)
      end if
      call eval_tree_evaluate (expect%lexpr)
      if (eval_tree_result_is_known (expect%lexpr)) then
        success = eval_tree_get_log (expect%lexpr)
        if (success) then
          call msg_message ("expect: success")
        else
          if (u >= 0) then
            call eval_tree_write (expect%lexpr, unit=u)
            flush (u)
          end if
          call msg_error ("expect: failure")
        end if
      else
        call msg_error ("expect: undefined result")
        success = .false.
      end if
      call expect_record (success)
      call rt_data_restore (global, expect%local)
    end subroutine cmd_expect_execute
```

Echo strings

⟨*Commands: types*⟩+≡
```
    type :: cmd_echo_t
      private
      type(eval_tree_t), dimension(:), allocatable :: expr
    end type cmd_echo_t
```

Finalize.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_echo_final (echo)
      type(cmd_echo_t), intent(inout) :: echo
      integer :: i
```

```
   if (allocated (echo%expr)) then
      do i = 1, size (echo%expr)
         call eval_tree_final (echo%expr(i))
      end do
   end if
end subroutine cmd_echo_final
```

Compile. Allocate an array which is filled with the names of the variables to echo.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_echo_compile (echo, pn, global)
  type(cmd_echo_t), pointer :: echo
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(in), target :: global
  type(parse_node_t), pointer :: pn_arg, pn_sexpr
  integer :: i, n_args
  pn_arg => parse_node_get_sub_ptr (pn, 2)
  allocate (echo)
  if (associated (pn_arg)) then
     n_args = parse_node_get_n_sub (pn_arg)
     allocate (echo%expr (n_args))
     pn_sexpr => parse_node_get_sub_ptr (pn_arg)
     i = 0
     do while (associated (pn_sexpr))
        i = i + 1
        call eval_tree_init_sexpr &
             (echo%expr(i), pn_sexpr, global%var_list)
        pn_sexpr => parse_node_get_next_ptr (pn_sexpr)
     end do
  else
     allocate (echo%expr (0))
  end if
end subroutine cmd_echo_compile
```

Execute. Scan the list of expressions, evaluate them and print the result.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_echo_execute (echo, global)
  type(cmd_echo_t), intent(inout) :: echo
  type(rt_data_t), intent(in), target :: global
  type(string_t) :: string
  integer :: i, u_out, u_log
  call msg_warning &
       ("The 'echo' command is deprecated.  Please use 'print/printf' instead.")
  u_out = output_unit ()
  u_log = logfile_unit (u_out)
  do i = 1, size (echo%expr)
     call eval_tree_evaluate (echo%expr(i))
     if (eval_tree_result_is_known (echo%expr(i))) then
        string = eval_tree_get_string (echo%expr(i))
        write (u_out, "(A)")  char (string)
        if (u_log >= 0)  write (u_log, "(A)")  char (string)
     end if
  end do
```

```
      flush (u_log)
    end subroutine cmd_echo_execute
```

Beams

The beam command includes both beam and structure-function definition. To
hold the structure functions, we define two special container types.

⟨*Commands: types*⟩+≡
```
    type :: strfun_def_t
       private
       integer :: type = STRF_NONE
       integer :: n_parameters = 1
       type(command_list_t), pointer :: options => null ()
       type(rt_data_t) :: local
    end type strfun_def_t

    type :: strfun_pair_t
       private
       integer :: n
       type(strfun_def_t), dimension(2) :: def
    end type strfun_pair_t

    type :: cmd_beams_t
       private
       integer :: n_in = 0
       type(eval_tree_t), dimension(:), allocatable :: pdg
       type(string_t), dimension(:), allocatable :: prt
       type(command_list_t), pointer :: options => null ()
       type(rt_data_t) :: local
       logical :: use_sqrts = .true.
       integer :: n_strfun = 0
       type(strfun_pair_t), dimension(:), allocatable :: strfun_pair
    end type cmd_beams_t
```

Delete the eval trees.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_beams_final (beams)
      type(cmd_beams_t), intent(inout) :: beams
      integer :: i
      do i = 1, beams%n_in
         call eval_tree_final (beams%pdg(i))
      end do
      if (associated (beams%options)) then
         call command_list_final (beams%options)
         deallocate (beams%options)
      end if
    end subroutine cmd_beams_final
```

Output

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_beams_write (beams, unit, indent)
```

```
      type(cmd_beams_t), intent(in) :: beams
      integer, intent(in), optional :: unit, indent
      integer :: u, i, ind
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)", advance="no")  "beams ("
      do i = 1, beams%n_in
         if (i /= 1)  write (u, "(', ')", advance="no")
         write (u, "(A)", advance="no")  char (beams%prt(i))
      end do
      write (u, "(A)", advance="no")  ")"
      if (associated (beams%options)) then
         write (u, "(1x,'{')")
         call command_list_write (beams%options, unit, indent)
         call write_indent (u, indent)
         write (u, "(1x,'}')")
      end if
      do i = 1, beams%n_strfun
         ind = 0;  if (present (indent))  ind = indent
         call strfun_pair_write (beams%strfun_pair(i), unit, indent=ind+3)
      end do
    end subroutine cmd_beams_write

    subroutine strfun_pair_write (strfun_pair, unit, indent)
      type(strfun_pair_t), intent(in) :: strfun_pair
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)", advance="no")  "=>"
      call strfun_def_write (strfun_pair%def(1), unit, indent)
      if (strfun_pair%n == 2) then
         write (u, "(1x,A)", advance="no")  ","
         call strfun_def_write (strfun_pair%def(2), unit, indent)
      end if
      write (u, *)
    end subroutine strfun_pair_write

    subroutine strfun_def_write (strfun_def, unit, indent)
      type(strfun_def_t), intent(in) :: strfun_def
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      select case (strfun_def%type)
      case (STRF_NONE);    write (u, "(1x,A)")  "none"
      case (STRF_LHAPDF);  write (u, "(1x,A)")  "lhapdf"
      case (STRF_ISR);     write (u, "(1x,A)")  "isr"
      case (STRF_EPA);     write (u, "(1x,A)")  "epa"
      case (STRF_EWA);     write (u, "(1x,A)")  "ewa"
      end select
      if (associated (strfun_def%options)) then
         write (u, "(1x,'{')")
         call command_list_write (strfun_def%options, unit, indent)
         call write_indent (u, indent)
```

```
         write (u, "(1x,'}')")
       end if
    end subroutine strfun_def_write
```

Compile.

⟨*Commands: procedures*⟩+≡

```
    subroutine cmd_beams_compile (beams, pn, global)
      type(cmd_beams_t), pointer :: beams
      type(parse_node_t), intent(in), target :: pn
      type(rt_data_t), intent(in), target :: global
      type(parse_node_t), pointer :: pn_beam_def, pn_beam_spec
      type(parse_node_t), pointer :: pn_beam_list, pn_opt
      type(parse_node_t), pointer :: pn_codes
      type(parse_node_t), pointer :: pn_strfun_seq, pn_strfun_pair
      integer :: i
      pn_beam_def => parse_node_get_sub_ptr (pn, 3)
      pn_beam_spec => parse_node_get_sub_ptr (pn_beam_def)
      pn_strfun_seq => parse_node_get_next_ptr (pn_beam_spec)
      pn_beam_list => parse_node_get_sub_ptr (pn_beam_spec)
      pn_opt => parse_node_get_next_ptr (pn_beam_list)
      allocate (beams)
      call rt_data_local_init (beams%local, global)
      if (associated (pn_opt)) then
         allocate (beams%options)
         call command_list_compile (beams%options, pn_opt, beams%local)
      end if
      beams%n_in = parse_node_get_n_sub (pn_beam_list)
      select case (beams%n_in)
      case (1)
         if (associated (pn_strfun_seq)) then
            call parse_node_write (pn_beam_def)
            call msg_error ("Structure functions can't be defined " &
                 // "for decay processes")
            pn_strfun_seq => null ()
         end if
      end select
      allocate (beams%pdg (beams%n_in))
      allocate (beams%prt (beams%n_in))
      pn_codes => parse_node_get_sub_ptr (pn_beam_list)
      do i = 1, beams%n_in
         call eval_tree_init_cexpr (beams%pdg(i), pn_codes, global%var_list)
         beams%prt(i) = "?"
         pn_codes => parse_node_get_next_ptr (pn_codes)
      end do
      if (associated (pn_strfun_seq)) then
         beams%n_strfun = parse_node_get_n_sub (pn_beam_def) - 1
         allocate (beams%strfun_pair (beams%n_strfun))
         do i = 1, beams%n_strfun
            pn_strfun_pair => parse_node_get_sub_ptr (pn_strfun_seq, 2)
            call strfun_pair_compile &
                 (beams%strfun_pair(i), pn_strfun_pair, beams%local)
            pn_strfun_seq => parse_node_get_next_ptr (pn_strfun_seq)
         end do
      end if
```

```
      end subroutine cmd_beams_compile

    subroutine strfun_pair_compile (strfun_pair, pn_strfun_pair, global)
      type(strfun_pair_t), intent(out) :: strfun_pair
      type(parse_node_t), intent(in), target :: pn_strfun_pair
      type(rt_data_t), intent(in), target :: global
      type(parse_node_t), pointer :: pn_strfun_def
      integer :: i
      strfun_pair%n = parse_node_get_n_sub (pn_strfun_pair)
      pn_strfun_def => parse_node_get_sub_ptr (pn_strfun_pair)
      do i = 1, strfun_pair%n
         call strfun_def_compile (strfun_pair%def(i), pn_strfun_def, global)
         pn_strfun_def => parse_node_get_next_ptr (pn_strfun_def)
      end do
    end subroutine strfun_pair_compile

    subroutine strfun_def_compile (strfun_def, pn_strfun_def, global)
      type(strfun_def_t), intent(out) :: strfun_def
      type(parse_node_t), intent(in), target :: pn_strfun_def
      type(rt_data_t), intent(in), target :: global
      type(parse_node_t), pointer :: pn_key, pn_opt
      pn_key => parse_node_get_sub_ptr (pn_strfun_def)
      pn_opt => parse_node_get_next_ptr (pn_key)
      select case (char (parse_node_get_key (pn_key)))
      case ("none")
         strfun_def%type = STRF_NONE
      case ("lhapdf")
         strfun_def%type = STRF_LHAPDF
      case ("isr")
         strfun_def%type = STRF_ISR
      case ("epa")
         strfun_def%type = STRF_EPA
      case ("ewa")
         strfun_def%type = STRF_EWA
      end select
      call rt_data_local_init (strfun_def%local, global)
      if (associated (pn_opt)) then
         allocate (strfun_def%options)
         call command_list_compile &
              (strfun_def%options, pn_opt, strfun_def%local)
      end if
    end subroutine strfun_def_compile
```

Command execution: Beam initialization. The output is a beam-data object,
which is copied to the global data block.

Structure functions are configured in the option section. They are linked to
the global data block, where they can be found by the integration command.

⟨*Commands: procedures*⟩+≡

```
    subroutine cmd_beams_execute (beams, global)
      type(cmd_beams_t), intent(inout), target :: beams
      type(rt_data_t), intent(inout), target :: global
      real(default) :: sqrts, p_cm, p_cm_theta, p_cm_phi
      type(pdg_array_t), dimension(2) :: aval
```

```
type(flavor_t), dimension(:), allocatable :: flv_tmp
type(flavor_t), dimension(2) :: flv
type(polarization_t), dimension(2) :: pol
integer :: i, u
u = logfile_unit ()
call lhapdf_status_reset (global%lhapdf_status)
call rt_data_link (beams%local, global)
if (associated (beams%options)) then
   call command_list_execute (beams%options, beams%local)
end if
do i = 1, beams%n_in
   call eval_tree_evaluate (beams%pdg(i))
   aval(i) = eval_tree_get_pdg_array (beams%pdg(i))
   call flavor_init (flv_tmp, aval(i), beams%local%model)
   select case (size (flv_tmp))
   case (1);  flv(i) = flv_tmp(1)
   case default
      call pdg_array_write (aval(i))
      call msg_fatal &
           ("Beam expression does not evaluate to a unique particle")
      return
   end select
   beams%prt(i) = flavor_get_name (flv(i))
   select case (beams%n_in)
   case (1)
      call polarization_init_trivial (pol(i), flv(i))
   case (2)
      call polarization_init_unpolarized (pol(i), flv(i))
   end select
end do
p_cm = var_list_get_rval (beams%local%var_list, var_str ("cm_momentum"))
p_cm_theta = &
     var_list_get_rval (beams%local%var_list, var_str ("cm_theta"))
p_cm_phi = &
     var_list_get_rval (beams%local%var_list, var_str ("cm_phi"))
select case (beams%n_in)
case (1)
   if (p_cm == 0 .and. p_cm_theta == 0) then
      call beam_data_init_decay (global%beam_data, flv, pol)
   else
      call beam_data_init_decay &
           (global%beam_data, flv, pol, p_cm, p_cm_theta, p_cm_phi)
   end if
case (2)
   if (beams%use_sqrts) then
      sqrts = var_list_get_rval (beams%local%var_list, var_str ("sqrts"))
      if (sqrts > 0) then
         if (p_cm == 0 .and. p_cm_theta == 0) then
            call beam_data_init_sqrts (global%beam_data, sqrts, flv, pol)
         else
            call beam_data_init_sqrts (global%beam_data, &
                 sqrts, flv, pol, p_cm, p_cm_theta, p_cm_phi)
         end if
      else
```

```
            call msg_fatal ("Beam setup: value of sqrts " &
                 // "must be set and positive")
            flush (u)
            call rt_data_restore (global, beams%local)
            return
         end if
      else
         call msg_bug ("Beam setup: individual beam setup not supported yet")
      end if
      if (global%sf_list_allocated) then
         call sf_list_final (global%sf_list)
         deallocate (global%sf_list)
      end if
      allocate (global%sf_list)
      global%sf_list_allocated = .true.
      do i = 1, beams%n_strfun
         call strfun_pair_register (beams%strfun_pair(i), global)
      end do
      call sf_list_freeze (global%sf_list)
      call sf_list_compute_md5sum (global%sf_list)
   end select
   call beam_data_write (global%beam_data, verbose=.false.)
   call beam_data_write (global%beam_data, verbose=.false., unit=u)
   flush (u)
   call rt_data_restore (global, beams%local)
end subroutine cmd_beams_execute

subroutine strfun_pair_register (strfun_pair, global)
  type(strfun_pair_t), intent(inout) :: strfun_pair
  type(rt_data_t), intent(inout), target :: global
  logical, dimension(2) :: affects_beam
  integer :: i
  select case (strfun_pair%n)
  case (1)
     affects_beam = .true.
     call strfun_def_register (strfun_pair%def(1), affects_beam, global)
  case (2)
     affects_beam = (/ .true., .false. /)
     call strfun_def_register (strfun_pair%def(1), affects_beam, global)
     affects_beam = (/ .false., .true. /)
     call strfun_def_register (strfun_pair%def(2), affects_beam, global)
  end select
end subroutine strfun_pair_register

subroutine strfun_def_register (strfun_def, affects_beam, global)
  type(strfun_def_t), intent(inout) :: strfun_def
  logical, dimension(2), intent(in) :: affects_beam
  type(rt_data_t), intent(inout), target :: global
  type(string_t) :: lhapdf_file
  type(sf_data_t), pointer :: sf_data
  integer :: lhapdf_member, lhapdf_photon_scheme
  real(default) :: isr_alpha, isr_q_max, isr_mass
  integer :: isr_order
  real(default) :: epa_alpha, epa_x_min, epa_q_min, epa_e_max, epa_mass
```

```
real(default) :: ewa_x_min, ewa_q_min, ewa_pt_max, ewa_mass, ewa_sqrts
logical :: ewa_keep_momentum, ewa_keep_energy
call rt_data_link (strfun_def%local, global)
if (associated (strfun_def%options)) then
   call command_list_execute (strfun_def%options, strfun_def%local)
end if
if (strfun_def%type /= STRF_NONE) then
   call sf_list_append (global%sf_list, &
        strfun_def%type, affects_beam, strfun_def%n_parameters, sf_data)
   select case (strfun_def%type)
   case (STRF_LHAPDF)
      lhapdf_file = var_list_get_sval (strfun_def%local%var_list, &
           var_str ("$lhapdf_file"))  ! $
      lhapdf_member = var_list_get_ival (strfun_def%local%var_list, &
           var_str ("lhapdf_member"))
      lhapdf_photon_scheme = var_list_get_ival (strfun_def%local%var_list, &
           var_str ("lhapdf_photon_scheme"))
      call sf_data_init_lhapdf (sf_data, global%lhapdf_status, &
           global%model, global%beam_data%flv, &
           lhapdf_file, lhapdf_member, lhapdf_photon_scheme)
   case (STRF_ISR)
      isr_alpha = var_list_get_rval (strfun_def%local%var_list, &
           var_str ("isr_alpha"))
      if (isr_alpha == 0) then
         isr_alpha = (var_list_get_rval (strfun_def%local%var_list, &
              var_str ("ee"))) ** 2 / (4 * pi)
      end if
      isr_q_max = var_list_get_rval (strfun_def%local%var_list, &
           var_str ("isr_q_max"))
      if (isr_q_max == 0) then
         isr_q_max = var_list_get_rval (strfun_def%local%var_list, &
              var_str ("sqrts"))
      end if
      isr_mass = var_list_get_rval (strfun_def%local%var_list, &
           var_str ("isr_mass"))
      isr_order = var_list_get_ival (strfun_def%local%var_list, &
           var_str ("isr_order"))
      if (isr_mass /= 0) then
         call sf_data_init_isr (sf_data, &
              global%model, global%beam_data%flv, &
              isr_alpha, isr_q_max, isr_mass, isr_order)
      else
         call sf_data_init_isr (sf_data, &
              global%model, global%beam_data%flv, &
              isr_alpha, isr_q_max, order = isr_order)
      end if
   case (STRF_EPA)
      epa_alpha = var_list_get_rval (strfun_def%local%var_list, &
           var_str ("epa_alpha"))
      if (epa_alpha == 0) then
         epa_alpha = (var_list_get_rval (strfun_def%local%var_list, &
              var_str ("ee"))) ** 2 / (4 * pi)
      end if
      epa_x_min = var_list_get_rval (strfun_def%local%var_list, &
```

```
               var_str ("epa_x_min"))
        epa_q_min = var_list_get_rval (strfun_def%local%var_list, &
               var_str ("epa_q_min"))
        epa_e_max = var_list_get_rval (strfun_def%local%var_list, &
               var_str ("epa_e_max"))
        if (epa_e_max == 0) then
           epa_e_max = var_list_get_rval (strfun_def%local%var_list, &
                  var_str ("sqrts"))
        end if
        epa_mass = var_list_get_rval (strfun_def%local%var_list, &
               var_str ("epa_mass"))
        if (epa_mass /= 0) then
           call sf_data_init_epa (sf_data, &
                  global%model, global%beam_data%flv, &
                  epa_alpha, epa_x_min, epa_q_min, epa_e_max, epa_mass)
        else
           call sf_data_init_epa (sf_data, &
                  global%model, global%beam_data%flv, &
                  epa_alpha, epa_x_min, epa_q_min, epa_e_max)
        end if
     case (STRF_EWA)
        call msg_warning ("EWA structure function not yet fully implemented")
        ewa_x_min = var_list_get_rval (strfun_def%local%var_list, &
               var_str ("ewa_x_min"))
        ewa_q_min = var_list_get_rval (strfun_def%local%var_list, &
               var_str ("ewa_q_min"))
        ewa_pt_max = var_list_get_rval (strfun_def%local%var_list, &
               var_str ("ewa_pt_max"))
        if (ewa_pt_max == 0) then
           ewa_pt_max = var_list_get_rval (strfun_def%local%var_list, &
                  var_str ("sqrts"))
        end if
        ewa_mass = var_list_get_rval (strfun_def%local%var_list, &
               var_str ("ewa_mass"))
        ewa_sqrts = var_list_get_rval (strfun_def%local%var_list, &
               var_str ("sqrts"))
        ewa_keep_momentum = var_list_get_lval (strfun_def%local%var_list, &
               var_str ("?ewa_keep_momentum"))
        ewa_keep_energy = var_list_get_lval (strfun_def%local%var_list, &
               var_str ("?ewa_keep_energy"))
        if (ewa_keep_momentum .and. ewa_keep_energy) &
           call msg_fatal (" EWA cannot violate both energy " &
                  // "and momentum conservation.")
        if (ewa_mass /= 0) then
           call sf_data_init_ewa (sf_data, &
                  global%model, global%beam_data%flv, &
                  ewa_x_min, ewa_q_min, ewa_pt_max, ewa_sqrts, &
                  ewa_keep_momentum, ewa_keep_energy, ewa_mass)
        else
           call sf_data_init_ewa (sf_data, &
                  global%model, global%beam_data%flv, &
                  ewa_x_min, ewa_q_min, ewa_pt_max, ewa_sqrts, &
                  ewa_keep_momentum, ewa_keep_energy)
        end if
```

```
      end select
    end if
    call rt_data_restore (global, strfun_def%local)
  end subroutine strfun_def_register
```

Cuts

Define a cut expression. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the process environment where the cut expression is used.

⟨*Commands: types*⟩+≡
```
    type :: cmd_cuts_t
       private
       type(parse_node_t), pointer :: pn_lexpr => null ()
    end type cmd_cuts_t
```

Output. Print the right-hand side as a parse tree.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_cuts_write (cuts, unit, indent)
      type(cmd_cuts_t), intent(in) :: cuts
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)")  "cuts ="
      call parse_node_write_rec (cuts%pn_lexpr, unit)
    end subroutine cmd_cuts_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_cuts_compile (cuts, pn)
      type(cmd_cuts_t), pointer :: cuts
      type(parse_node_t), intent(in), target :: pn
      allocate (cuts)
      cuts%pn_lexpr => parse_node_get_sub_ptr (pn, 3)
    end subroutine cmd_cuts_compile
```

Instead of evaluating the cut expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_cuts_execute (cuts, global)
      type(cmd_cuts_t), intent(inout), target :: cuts
      type(rt_data_t), intent(inout), target :: global
      global%pn_cuts_lexpr => cuts%pn_lexpr
    end subroutine cmd_cuts_execute
```

Weight

Define a weight expression. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the process environment

1221

where the expression is used.

⟨*Commands: types*⟩+≡
```
    type :: cmd_weight_t
       private
       type(parse_node_t), pointer :: pn_expr => null ()
    end type cmd_weight_t
```

Output. Print the right-hand side as a parse tree.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_weight_write (weight, unit, indent)
      type(cmd_weight_t), intent(in) :: weight
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)")  "weight ="
      call parse_node_write_rec (weight%pn_expr, unit)
    end subroutine cmd_weight_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_weight_compile (weight, pn)
      type(cmd_weight_t), pointer :: weight
      type(parse_node_t), intent(in), target :: pn
      allocate (weight)
      weight%pn_expr => parse_node_get_sub_ptr (pn, 3)
    end subroutine cmd_weight_compile
```

Instead of evaluating the expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_weight_execute (weight, global)
      type(cmd_weight_t), intent(inout), target :: weight
      type(rt_data_t), intent(inout), target :: global
      global%pn_weight_expr => weight%pn_expr
    end subroutine cmd_weight_execute
```

Scale

Define a scale expression. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the process environment where the expression is used.

⟨*Commands: types*⟩+≡
```
    type :: cmd_scale_t
       private
       type(parse_node_t), pointer :: pn_expr => null ()
    end type cmd_scale_t
```

Output. Print the right-hand side as a parse tree.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_scale_write (scale, unit, indent)
  type(cmd_scale_t), intent(in) :: scale
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "scale ="
  call parse_node_write_rec (scale%pn_expr, unit)
end subroutine cmd_scale_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_scale_compile (scale, pn)
  type(cmd_scale_t), pointer :: scale
  type(parse_node_t), intent(in), target :: pn
  allocate (scale)
  scale%pn_expr => parse_node_get_sub_ptr (pn, 3)
end subroutine cmd_scale_compile
```

Instead of evaluating the scale expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_scale_execute (scale, global)
  type(cmd_scale_t), intent(inout), target :: scale
  type(rt_data_t), intent(inout), target :: global
  global%pn_scale_expr => scale%pn_expr
end subroutine cmd_scale_execute
```

Integration

Integrate several processes, consecutively with identical parameters.

⟨*Commands: types*⟩+≡

```
type :: cmd_integrate_t
  private
  integer :: n_proc = 0
  type(string_t), dimension(:), allocatable :: process_id
  type(grid_parameters_t) :: grid_parameters
  type(phs_parameters_t) :: phs_par
  type(mapping_defaults_t) :: mapping_defaults
  type(command_list_t), pointer :: options => null ()
  type(rt_data_t) :: local
end type cmd_integrate_t
```

Finalizer.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_integrate_final (integrate)
  type(cmd_integrate_t), intent(inout) :: integrate
  if (associated (integrate%options)) then
```

1223

```
          call command_list_final (integrate%options)
          deallocate (integrate%options)
       end if
    end subroutine cmd_integrate_final
```

Output.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_integrate_write (integrate, unit, indent)
      type(cmd_integrate_t), intent(in) :: integrate
      integer, intent(in), optional :: unit, indent
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)", advance="no") "integrate ("
      do i = 1, integrate%n_proc
         if (i /= 1)  write (u, "(', ')", advance="no")
         write (u, "(A)", advance="no")  char (integrate%process_id(i))
      end do
      write (u, "(A)") ")"
      if (associated (integrate%options)) then
         write (u, "(1x,'{')")
         call command_list_write (integrate%options, unit, indent)
         call write_indent (u, indent)
         write (u, "(1x,'}')")
      end if
    end subroutine cmd_integrate_write
```

Compile.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_integrate_compile (integrate, pn, global)
      type(cmd_integrate_t), pointer :: integrate
      type(parse_node_t), intent(in), target :: pn
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_proclist, pn_proc, pn_opt
      integer :: i
      pn_proclist => parse_node_get_sub_ptr (pn, 2)
      pn_opt => parse_node_get_next_ptr (pn_proclist)
      allocate (integrate)
      call rt_data_local_init (integrate%local, global)
      if (associated (pn_opt)) then
         allocate (integrate%options)
         call command_list_compile (integrate%options, pn_opt, integrate%local)
      end if
      integrate%n_proc = parse_node_get_n_sub (pn_proclist)
      allocate (integrate%process_id (integrate%n_proc))
      pn_proc => parse_node_get_sub_ptr (pn_proclist)
      do i = 1, integrate%n_proc
         integrate%process_id(i) = parse_node_get_string (pn_proc)
         call var_list_init_process_results (global%var_list, &
              integrate%process_id (i))
         pn_proc => parse_node_get_next_ptr (pn_proc)
      end do
      ! API for setting grid_parameters etc. missing
```

1224

```
      end subroutine cmd_integrate_compile
```

Command execution. Integrate the process with the predefined number of
passes, iterations and calls. For structure functions, cuts, weight and scale,
use local definitions if present; by default, the local definitions are initialized
with the global ones.

NOTE: The process must be linked to the global variable list; the local list
is deleted when the `cmd_integrate` object is deleted, but the process object
exists independent of this. (This matters in interactive mode only.) We should
look for a robust implementation of local variables that avoids this problem.

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_integrate_execute (integrate, global)
     type(cmd_integrate_t), intent(inout), target :: integrate
     type(rt_data_t), intent(inout), target :: global
     logical :: use_beams
     type(process_t), pointer :: process
     integer :: proc, n_out, pass, i, n_calls
     type(iterations_list_t), dimension(:), allocatable :: it_list
     type(iterations_spec_t) :: it_spec
     logical :: ok, use_default_iterations, rebuild_phs, phs_only, rebuild_grids
     type(string_t) :: phs_filename, grids_filename
     logical :: hs_active
     real(default) :: hs_threshold
     integer :: hs_cutoff
     real(default) :: alpha_s, sqrts
     character(32) :: md5sum_beams, md5sum_sf_list, md5sum_mappings
     character(32) :: md5sum_cuts, md5sum_weight, md5sum_scale
     integer :: current_pass, current_it, it
     logical :: adapt_final_grids, adapt_final_weights
     logical :: time_estimate
     integer :: u, u_tmp
     u = logfile_unit ()
     call rt_data_link (integrate%local, global)
     if (associated (integrate%options)) then
        call command_list_execute (integrate%options, integrate%local)
     end if
     call maybe_cmd_compile_execute (integrate%process_id, integrate%local)
     integrate%grid_parameters%threshold_calls = &
          var_list_get_ival (integrate%local%var_list, &
                             var_str ("threshold_calls"))
     integrate%grid_parameters%min_calls_per_channel = &
          var_list_get_ival (integrate%local%var_list, &
                             var_str ("min_calls_per_channel"))
     integrate%grid_parameters%min_calls_per_bin = &
          var_list_get_ival (integrate%local%var_list, &
                             var_str ("min_calls_per_bin"))
     integrate%grid_parameters%min_bins = &
          var_list_get_ival (integrate%local%var_list, &
                             var_str ("min_bins"))
     integrate%grid_parameters%max_bins = &
          var_list_get_ival (integrate%local%var_list, &
                             var_str ("max_bins"))
     integrate%grid_parameters%stratified = &
```

```
      var_list_get_lval (integrate%local%var_list, &
                        var_str ("?stratified"))
integrate%grid_parameters%use_vamp_equivalences = &
      var_list_get_lval (integrate%local%var_list, &
                        var_str ("?use_vamp_equivalences"))
integrate%grid_parameters%channel_weights_power = &
      var_list_get_rval (integrate%local%var_list, &
                        var_str ("channel_weights_power"))
integrate%phs_par%m_threshold_s = &
      var_list_get_rval (integrate%local%var_list, &
                        var_str ("phs_threshold_s"))
integrate%phs_par%m_threshold_t = &
      var_list_get_rval (integrate%local%var_list, &
                        var_str ("phs_threshold_t"))
integrate%phs_par%off_shell = &
      var_list_get_ival (integrate%local%var_list, &
                        var_str ("phs_off_shell"))
integrate%phs_par%t_channel = &
      var_list_get_ival (integrate%local%var_list, &
                        var_str ("phs_t_channel"))
use_beams = beam_data_are_valid (integrate%local%beam_data)
if (use_beams) then
   if (.not. beam_data_masses_are_consistent &
                (integrate%local%beam_data)) then
      call msg_warning &
            ("Masses of beam particle(s) differ from beam masses")
   end if
end if

allocate (it_list (integrate%n_proc))
it_list = integrate%local%it_list

LOOP_PROC: do proc = 1, integrate%n_proc
   call msg_message ("Integrating process '" // &
        char (integrate%process_id(proc)) // "'")
   call process_store_init_process (process, &
        integrate%local%prc_lib, &
        integrate%process_id(proc), integrate%local%model, &
        global%lhapdf_status, &
        integrate%local%var_list, use_beams=use_beams)
   if (.not. process_is_valid (process)) then
      call msg_fatal ("Integrating process '" &
           // char (integrate%process_id(proc)) // "': " &
           // "initialization failed, skipping")
      cycle LOOP_PROC
   else if (.not. process_has_matrix_element (process)) then
      call process_results_write_header (process, logfile=.false.)
      call process_results_write_header (process, unit=u)
      call process_do_dummy_integration (process)
      call process_results_write_footer (process, no_line=.true.)
      call process_results_write_footer (process, unit=u, no_line=.true.)
      flush (u)
      call process_record_integral (process, global%var_list)
      cycle LOOP_PROC
```

```
end if
sqrts = var_list_get_rval (integrate%local%var_list, var_str ("sqrts"))
md5sum_beams = beam_data_get_md5sum (integrate%local%beam_data, sqrts)
md5sum_sf_list = ""
if (use_beams) then
   if (beam_data_get_n_in (integrate%local%beam_data) &
        /= process_get_n_in (process)) then
      call msg_fatal ("Process '" // char (integrate%process_id(proc)) &
           // "': beam/process mismatch (collision/decay)", &
           (/ var_str ("   ----------------------------------------"), &
              var_str ("This possibly means that you tried to generate "), &
              var_str ("a forbidden process, for which WHIZARD could not"), &
              var_str ("find a valid phase space channel. Or there is a"), &
              var_str ("mismatch between beams and hard interaction.") /) )
      return
   end if
   if (associated (integrate%local%sf_list)) then
      md5sum_sf_list = sf_list_get_md5sum (integrate%local%sf_list)
      call process_setup_beams (process, integrate%local%beam_data, &
           sf_list_get_n_strfun (integrate%local%sf_list), &
           sf_list_get_n_mapping (integrate%local%sf_list))
      call process_check_beam_setup (process, integrate%local%var_list)
      call process_setup_strfun (process, integrate%local%sf_list)
   else
      call process_setup_beams (process, integrate%local%beam_data, 0, 0)
   end if
else
   call process_setup_beams (process, integrate%local%beam_data, 0, 0, &
        sqrts = sqrts)
end if
call process_connect_strfun (process, ok)
if (.not. ok) then
   call msg_error ("Process '" // char (integrate%process_id(proc)) &
        // "': beam/structure function setup failed, skipped")
   cycle LOOP_PROC
end if
rebuild_phs = var_list_get_lval (integrate%local%var_list, &
     var_str ("?rebuild_phase_space"))
phs_filename = var_list_get_sval (integrate%local%var_list, &
     var_str ("$phs_file"))   ! $ fool the noweb emacs mode
phs_only = var_list_get_lval (integrate%local%var_list, &
     var_str ("?phs_only"))
integrate%mapping_defaults%energy_scale = &
     var_list_get_rval (integrate%local%var_list, &
     var_str ("phs_e_scale"))
integrate%mapping_defaults%invariant_mass_scale = &
     var_list_get_rval (integrate%local%var_list, &
     var_str ("phs_m_scale"))
integrate%mapping_defaults%momentum_transfer_scale = &
     var_list_get_rval (integrate%local%var_list, &
     var_str ("phs_q_scale"))
md5sum_mappings = &
     mapping_defaults_md5sum (integrate%mapping_defaults)
if (phs_filename == "") then
```

```
      call process_setup_phase_space (process, rebuild_phs, &
           integrate%phs_par, integrate%mapping_defaults, &
           filename_out = integrate%process_id(proc) // ".phs", &
           ok = ok)
   else
      call process_setup_phase_space (process, rebuild_phs, &
           integrate%phs_par, integrate%mapping_defaults, &
           filename_in = phs_filename, &
           filename_out = integrate%process_id(proc) // ".phs", &
           ok = ok)
   end if
   if (.not. ok) then
      call msg_error ("Process '" // char (integrate%process_id(proc)) &
           // "': phase space setup failed, skipped")
      cycle LOOP_PROC
   end if
   if (phs_only) then
      call msg_message ("Process '" // char (integrate%process_id(proc)) &
           // "': phase space setup complete.")
      cycle LOOP_PROC
   end if
   if (associated (integrate%local%pn_cuts_lexpr)) then
      call process_setup_cuts (process, integrate%local%pn_cuts_lexpr)
      md5sum_cuts = parse_node_get_md5sum (integrate%local%pn_cuts_lexpr)
      call msg_message ("Applying user-defined cuts.")
   else
      md5sum_cuts = ""
      call msg_warning ("No cuts have been defined.")
   end if
   if (associated (integrate%local%pn_weight_expr)) then
      call process_setup_weight (process, integrate%local%pn_weight_expr)
      md5sum_weight = parse_node_get_md5sum (integrate%local%pn_weight_expr)
      call msg_message ("Using user-defined integration weight.")
   else
      md5sum_weight = ""
   end if
   if (associated (integrate%local%pn_scale_expr)) then
      call process_setup_scale (process, integrate%local%pn_scale_expr)
      md5sum_scale = parse_node_get_md5sum (integrate%local%pn_scale_expr)
      call msg_message ("Using user-defined event scale setup.")
   else
      md5sum_scale = ""
      call msg_message ("Using partonic energy as event scale.")
   end if
   if (var_list_is_known (integrate%local%var_list, &
        var_str ("alphas"))) then
      alpha_s = var_list_get_rval (integrate%local%var_list, &
           var_str ("alphas"))
      call process_set_alpha_s (process, alpha_s)
   end if

   adapt_final_grids = var_list_get_lval (integrate%local%var_list, &
        var_str ("?adapt_final_grids"))
   adapt_final_weights = var_list_get_lval (integrate%local%var_list, &
```

```
      var_str ("?adapt_final_weights"))

time_estimate = var_list_get_lval (integrate%local%var_list, &
     var_str ("?time_estimate"))

n_out = process_get_n_out (process)
use_default_iterations = it_list(proc)%n_pass == 0
if (use_default_iterations) then
   select case (n_out)
   case (:1)
      call msg_error ("Integrate: number of outgoing particles " &
           // "must be at least 2")
      cycle LOOP_PROC
   case (2:ITERATIONS_DEFAULT_LIST_SIZE)
      it_list(proc) = integrate%local%it_list_default(n_out)
   case default
      it_list(proc) = &
           integrate%local%it_list_default(ITERATIONS_DEFAULT_LIST_SIZE)
   end select
else
   call iterations_list_complete (it_list(proc), &
        integrate%local%it_list_default(n_out))
end if
call iterations_list_adjust_n_calls (it_list(proc), &
     process, integrate%grid_parameters)
call iterations_list_write (it_list(proc))
call process_init_vamp_history &
     (process, iterations_list_get_n_it (it_list(proc)))

if (it_list(proc)%n_pass > 0) then
   it_spec = it_list(proc)%pass(1)
   n_calls = it_spec%n_calls
   grids_filename = process_get_id (process) // ".vg"
   rebuild_grids = var_list_get_lval (integrate%local%var_list, &
        var_str ("?rebuild_grids"))
   if (.not. rebuild_grids) then
      call process_read_grid_file (process, &
           grids_filename, &
           md5sum_beams, md5sum_sf_list, md5sum_mappings, &
           md5sum_cuts, md5sum_weight, md5sum_scale, &
           integrate%grid_parameters, &
           iterations_list_get_pass_array (it_list(proc)), &
           iterations_list_get_n_calls_array (it_list(proc)),&
           ok)
      rebuild_grids = .not. ok
   end if
   if (rebuild_grids) then
      call process_setup_grids &
           (process, integrate%grid_parameters, calls=n_calls)
      write (msg_buffer, "(4(I0,A),A,L1)")  &
           n_calls, " calls, ", &
           process_get_n_channels (process), " channels, ", &
           process_get_n_parameters (process), " dimensions, ", &
           process_get_n_bins (process), " bins, ", &
```

```
                "stratified = ", integrate%grid_parameters%stratified
        call msg_message ()
     else
        write (msg_buffer, "(3(I0,A),A,L1)")  &
             n_calls, " calls, ", &
             process_get_n_channels (process), " channels, ", &
             process_get_n_parameters (process), " dimensions, ", &
             "stratified = ", integrate%grid_parameters%stratified
        call msg_message ()
     end if
     current_pass = process_get_current_pass (process)
     current_it = process_get_current_it (process)
  end if

  hs_active = var_list_get_lval (integrate%local%var_list, &
       var_str ("?helicity_selection_active"))
  if (hs_active) then
     hs_threshold = var_list_get_rval (integrate%local%var_list, &
          var_str ("helicity_selection_threshold"))
     hs_cutoff = var_list_get_ival (integrate%local%var_list, &
          var_str ("helicity_selection_cutoff"))
  else
     hs_threshold = -1
     hs_cutoff = 1000
  end if
  call process_reset_helicity_selection &
       (process, hs_threshold, hs_cutoff)

  call process_results_write_header (process, logfile=.false.)
  call process_results_write_header (process, unit=u)
  flush (u)
  it = 0
  LOOP_PASS: do pass = 1, it_list(proc)%n_pass - 1
     it_spec = it_list(proc)%pass(pass)
     n_calls = it_spec%n_calls
     LOOP_IT: do i = 1, it_spec%n_it
        it = it + 1
        if (pass < current_pass &
             .or. pass == current_pass .and. i <= current_it) then
           call process_results_write_entry (process, it)
           call process_results_write_entry (process, it, unit=u)
           flush (u)
        else
           call process_integrate (process, &
                integrate%local%rng, integrate%grid_parameters, &
                pass, 1, 1, n_calls, &
                i==1, .true., i>2, .true., &
                time_estimate, &
                grids_filename, &
                md5sum_beams, md5sum_sf_list, md5sum_mappings, &
                md5sum_cuts, md5sum_weight, md5sum_scale)
        end if
     end do LOOP_IT
     call process_results_write_average (process, pass)
```

1230

```
          call process_results_write_average (process, pass, unit=u)
          flush (u)
          call process_record_integral (process, global%var_list)
          call process_write_logfile (process)
       end do LOOP_PASS

       pass = it_list(proc)%n_pass
       if (pass > current_pass)  current_it = 0
       if (pass > 0 .and. pass >= current_pass) then
          it_spec = it_list(proc)%pass(pass)
          n_calls = it_spec%n_calls
          do i = 1, current_it
             it = it + 1
             call process_results_write_entry (process, it)
             call process_results_write_entry (process, it, unit=u)
             flush (u)
          end do
          call process_integrate (process, &
               integrate%local%rng, integrate%grid_parameters, &
               pass, current_it+1, it_spec%n_it, n_calls, &
               .true., adapt_final_grids, adapt_final_weights, .true., &
               time_estimate, &
               grids_filename, &
               md5sum_beams, md5sum_sf_list, md5sum_mappings, &
               md5sum_cuts, md5sum_weight, md5sum_scale)
          call process_record_integral (process, global%var_list)
          call process_write_logfile (process)
       end if
       call process_results_write_footer (process)
       call process_results_write_footer (process, unit=u)
       if (time_estimate)  call process_write_time_estimate (process)
       flush (u)
    end do LOOP_PROC

    call rt_data_restore (global, integrate%local)
  end subroutine cmd_integrate_execute
```

If the relevant process library is not completely compiled and loaded, we silently insert a 'compile' command.

⟨*Commands: procedures*⟩+≡

```
    subroutine maybe_cmd_compile_execute (process_id, global)
      type(string_t), dimension(:), intent(in) :: process_id
      type(rt_data_t), intent(inout), target :: global
      type(cmd_compile_t), pointer :: compile
      integer :: i
      if (associated (global%prc_lib)) then
         call process_library_update_status (global%prc_lib)
         if (.not. process_library_is_compiled (global%prc_lib)) then
            allocate (compile)
            allocate (compile%libname (1))
            compile%libname(1) = process_library_get_name (global%prc_lib)
            call cmd_compile_execute (compile, global)
            deallocate (compile)
```

```
         end if
      else
         call msg_bug ("Integrate: no process library active")
      end if
   end subroutine maybe_cmd_compile_execute
```

Observables

Declare an observable. After the declaration, it can be used to record data, and
at the end one can retrieve average and error.

⟨*Commands: types*⟩+≡
```
   type :: cmd_observable_t
      private
      logical :: use_id_expr = .false.
      type(string_t) :: id
      type(eval_tree_t) :: expr_id
      type(command_list_t), pointer :: options => null ()
      type(rt_data_t) :: local
   end type cmd_observable_t
```

Finalizer for the ID string expression

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_observable_final (observable)
      type(cmd_observable_t), intent(inout) :: observable
      call eval_tree_final (observable%expr_id)
      if (associated (observable%options)) then
         call command_list_final (observable%options)
         deallocate (observable%options)
      end if
   end subroutine cmd_observable_final
```

Output.

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_observable_write (observable, unit, indent)
      type(cmd_observable_t), intent(in) :: observable
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)", advance="no") "observable"
      if (observable%use_id_expr) then
         call eval_tree_write (observable%expr_id, unit)
      else
         write (u, "(1x,A)", advance="no")  char (observable%id)
      end if
      write (u, *)
   end subroutine cmd_observable_write
```

Compile. Just record the observable ID.

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_observable_compile (observable, pn, global)
```

```
    type(cmd_observable_t), pointer :: observable
    type(parse_node_t), intent(in), target :: pn
    type(rt_data_t), intent(in), target :: global
    type(parse_node_t), pointer :: pn_tag, pn_opt
    pn_tag => parse_node_get_sub_ptr (pn, 2)
    if (associated (pn_tag)) then
       pn_opt => parse_node_get_next_ptr (pn_tag)
    else
       pn_opt => null ()
    end if
    allocate (observable)
    call rt_data_local_init (observable%local, global)
    if (associated (pn_opt)) then
       allocate (observable%options)
       call command_list_compile (observable%options, pn_opt, observable%local)
    end if
    select case (char (parse_node_get_rule_key (pn_tag)))
    case ("analysis_id")
       observable%id = parse_node_get_string (pn_tag)
    case default
       observable%use_id_expr = .true.
       call eval_tree_init_sexpr &
            (observable%expr_id, pn_tag, observable%local%var_list)
    end select
  end subroutine cmd_observable_compile
```

Command execution. This declares the observable and allocates it in the analysis store.

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_observable_execute (observable, global)
    type(cmd_observable_t), intent(inout), target :: observable
    type(rt_data_t), intent(inout), target :: global
    type(string_t) :: label, physical_unit, title
    type(plot_labels_t) :: plot_labels
    call rt_data_link (observable%local, global)
    if (associated (observable%options)) then
       call command_list_execute (observable%options, observable%local)
    end if
    if (observable%use_id_expr) then
       call eval_tree_evaluate (observable%expr_id)
       observable%id = eval_tree_get_string (observable%expr_id)
    end if
    label = var_list_get_sval (observable%local%var_list, var_str ("$label"))
    physical_unit = var_list_get_sval &
         (observable%local%var_list, var_str ("$physical_unit"))
    title = var_list_get_sval (observable%local%var_list, var_str ("$title"))
    call plot_labels_init (plot_labels, title)
    call analysis_init_observable &
         (observable%id, label, physical_unit, plot_labels)
    call rt_data_restore (global, observable%local)
  end subroutine cmd_observable_execute
```

Histograms

Declare a histogram. At minimum, we have to set lower and upper bound and bin width.

⟨*Commands: types*⟩+≡
```
type :: cmd_histogram_t
  private
  type(string_t) :: id
  logical :: use_id_expr = .false.
  type(eval_tree_t) :: expr_id
  type(eval_tree_t) :: expr_lower_bound
  type(eval_tree_t) :: expr_upper_bound
  type(eval_tree_t) :: expr_bin_width
  type(command_list_t), pointer :: options => null ()
  type(rt_data_t) :: local
end type cmd_histogram_t
```

Finalizer for the eval trees.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_histogram_final (histogram)
  type(cmd_histogram_t), intent(inout) :: histogram
  call eval_tree_final (histogram%expr_id)
  call eval_tree_final (histogram%expr_lower_bound)
  call eval_tree_final (histogram%expr_upper_bound)
  call eval_tree_final (histogram%expr_bin_width)
  if (associated (histogram%options)) then
     call command_list_final (histogram%options)
     deallocate (histogram%options)
  end if
end subroutine cmd_histogram_final
```

Output.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_histogram_write (histogram, unit, indent)
  type(cmd_histogram_t), intent(in) :: histogram
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)", advance="no") "histogram"
  if (histogram%use_id_expr) then
     call eval_tree_write (histogram%expr_id, unit)
  else
     write (u, "(1x,A)", advance="no")  char (histogram%id)
  end if
  write (u, "(1x,A)") "("
  call eval_tree_write (histogram%expr_lower_bound, unit)
  call write_indent (u, indent)
  write (u, "(1x,A)") ","
  call eval_tree_write (histogram%expr_upper_bound, unit)
  call write_indent (u, indent)
  write (u, "(1x,A)") ","
  call eval_tree_write (histogram%expr_bin_width, unit)
```

```
      call write_indent (u, indent)
      write (u, "(1x,A)") ")"
    end subroutine cmd_histogram_write
```

Compile. Record the histogram ID and initialize lower, upper bound and bin width.

⟨*Commands: procedures*⟩+≡

```
    subroutine cmd_histogram_compile (histogram, pn, global)
      type(cmd_histogram_t), pointer :: histogram
      type(parse_node_t), intent(in), target :: pn
      type(rt_data_t), intent(in), target :: global
      type(parse_node_t), pointer :: pn_tag, pn_args, pn_arg1, pn_arg2, pn_arg3
      type(parse_node_t), pointer :: pn_opt
      pn_tag => parse_node_get_sub_ptr (pn, 2)
      pn_args => parse_node_get_next_ptr (pn_tag)
      pn_arg1 => parse_node_get_sub_ptr (pn_args)
      pn_arg2 => parse_node_get_next_ptr (pn_arg1)
      pn_arg3 => parse_node_get_next_ptr (pn_arg2)
      if (associated (pn_args)) then
         pn_opt => parse_node_get_next_ptr (pn_args)
      else
         pn_opt => null ()
      end if
      allocate (histogram)
      call rt_data_local_init (histogram%local, global)
      if (associated (pn_opt)) then
         allocate (histogram%options)
         call command_list_compile (histogram%options, pn_opt, histogram%local)
      end if
      select case (char (parse_node_get_rule_key (pn_tag)))
      case ("analysis_id")
         histogram%id = parse_node_get_string (pn_tag)
      case default
         histogram%use_id_expr = .true.
         call eval_tree_init_sexpr &
              (histogram%expr_id, pn_tag, histogram%local%var_list)
      end select
      call eval_tree_init_expr &
           (histogram%expr_lower_bound, pn_arg1, histogram%local%var_list)
      call eval_tree_init_expr &
           (histogram%expr_upper_bound, pn_arg2, histogram%local%var_list)
      call eval_tree_init_expr &
           (histogram%expr_bin_width, pn_arg3, histogram%local%var_list)
    end subroutine cmd_histogram_compile
```

Command execution. This declares the histogram and allocates it in the analysis store.

⟨*Commands: procedures*⟩+≡

```
    subroutine cmd_histogram_execute (histogram, global)
      type(cmd_histogram_t), intent(inout), target :: histogram
      type(rt_data_t), intent(inout), target :: global
      real(default) :: lower_bound, upper_bound, bin_width
      logical :: bounds_are_known
```

```
type(string_t) :: label, physical_unit
type(string_t) :: title, description, xlabel, ylabel
type(plot_labels_t) :: plot_labels
call rt_data_link (histogram%local, global)
if (associated (histogram%options)) then
   call command_list_execute (histogram%options, histogram%local)
end if
if (histogram%use_id_expr) then
   call eval_tree_evaluate (histogram%expr_id)
   histogram%id = eval_tree_get_string (histogram%expr_id)
end if
bounds_are_known = .true.
call eval_tree_evaluate (histogram%expr_lower_bound)
call eval_tree_evaluate (histogram%expr_upper_bound)
call eval_tree_evaluate (histogram%expr_bin_width)
if (eval_tree_result_is_known (histogram%expr_lower_bound)) then
   lower_bound = eval_tree_get_real (histogram%expr_lower_bound)
else
   call msg_error ("Histogram '" &
        // char (histogram%id) // "': lower bound is undefined")
   bounds_are_known = .false.
end if
if (eval_tree_result_is_known (histogram%expr_upper_bound)) then
   upper_bound = eval_tree_get_real (histogram%expr_upper_bound)
else
   call msg_error ("Histogram '" &
        // char (histogram%id) // "': upper bound is undefined")
   bounds_are_known = .false.
end if
if (eval_tree_result_is_known (histogram%expr_bin_width)) then
   bin_width = eval_tree_get_real (histogram%expr_bin_width)
else
   call msg_error ("Histogram '" &
        // char (histogram%id) // "': bin width is undefined")
   bounds_are_known = .false.
end if
label = var_list_get_sval (histogram%local%var_list, var_str ("$label"))
physical_unit = var_list_get_sval &
     (histogram%local%var_list, var_str ("$physical_unit"))
title = var_list_get_sval (histogram%local%var_list, var_str ("$title"))
description = &
     var_list_get_sval (histogram%local%var_list, var_str ("$description"))
xlabel = var_list_get_sval (histogram%local%var_list, var_str ("$xlabel"))
ylabel = var_list_get_sval (histogram%local%var_list, var_str ("$ylabel"))
call plot_labels_init (plot_labels, title, &
     description, xlabel, ylabel)
if (bounds_are_known) then
   call analysis_init_histogram &
        (histogram%id, lower_bound, upper_bound, bin_width, &
         label, physical_unit, plot_labels)
else
   call msg_error ("Histogram '" &
        // char (histogram%id) // "': invalid declaration, skipping")
end if
```

```
      call rt_data_restore (global, histogram%local)
    end subroutine cmd_histogram_execute
```

Plots

Declare a plot. At minimum, we have to set lower and upper bound and bin width.

⟨*Commands: types*⟩+≡

```
    type :: cmd_plot_t
       private
       type(string_t) :: id
       logical :: use_id_expr = .false.
       type(eval_tree_t) :: expr_id
       type(eval_tree_t) :: expr_lower_bound
       type(eval_tree_t) :: expr_upper_bound
       type(command_list_t), pointer :: options => null ()
       type(rt_data_t) :: local
    end type cmd_plot_t
```

Finalizer for the eval trees.

⟨*Commands: procedures*⟩+≡

```
    subroutine cmd_plot_final (plot)
      type(cmd_plot_t), intent(inout) :: plot
      call eval_tree_final (plot%expr_id)
      call eval_tree_final (plot%expr_lower_bound)
      call eval_tree_final (plot%expr_upper_bound)
      if (associated (plot%options)) then
         call command_list_final (plot%options)
         deallocate (plot%options)
      end if
    end subroutine cmd_plot_final
```

Output.

⟨*Commands: procedures*⟩+≡

```
    subroutine cmd_plot_write (plot, unit, indent)
      type(cmd_plot_t), intent(in) :: plot
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)", advance="no") "plot"
      if (plot%use_id_expr) then
         call eval_tree_write (plot%expr_id, unit)
      else
         write (u, "(1x,A)", advance="no")  char (plot%id)
      end if
      write (u, "(1x,A)") "("
      call eval_tree_write (plot%expr_lower_bound, unit)
      call write_indent (u, indent)
      write (u, "(1x,A)") ","
      call eval_tree_write (plot%expr_upper_bound, unit)
```

```
          call write_indent (u, indent)
          write (u, "(1x,A)") ")"
        end subroutine cmd_plot_write
```

Compile. Record the plot ID and initialize lower, upper bound and bin width.

⟨*Commands: procedures*⟩+≡
```
      subroutine cmd_plot_compile (plot, pn, global)
        type(cmd_plot_t), pointer :: plot
        type(parse_node_t), intent(in), target :: pn
        type(rt_data_t), intent(in), target :: global
        type(parse_node_t), pointer :: pn_tag, pn_args, pn_arg1, pn_arg2
        type(parse_node_t), pointer :: pn_opt
        pn_tag => parse_node_get_sub_ptr (pn, 2)
        pn_args => parse_node_get_next_ptr (pn_tag)
        pn_arg1 => parse_node_get_sub_ptr (pn_args)
        pn_arg2 => parse_node_get_next_ptr (pn_arg1)
        if (associated (pn_args)) then
           pn_opt => parse_node_get_next_ptr (pn_args)
        else
           pn_opt => null ()
        end if
        allocate (plot)
        call rt_data_local_init (plot%local, global)
        if (associated (pn_opt)) then
           allocate (plot%options)
           call command_list_compile (plot%options, pn_opt, plot%local)
        end if
        select case (char (parse_node_get_rule_key (pn_tag)))
        case ("analysis_id")
           plot%id = parse_node_get_string (pn_tag)
        case default
           plot%use_id_expr = .true.
           call eval_tree_init_sexpr (plot%expr_id, pn_tag, plot%local%var_list)
        end select
        call eval_tree_init_expr &
             (plot%expr_lower_bound, pn_arg1, plot%local%var_list)
        call eval_tree_init_expr &
             (plot%expr_upper_bound, pn_arg2, plot%local%var_list)
      end subroutine cmd_plot_compile
```

Command execution. This declares the plot and allocates it in the analysis store.

⟨*Commands: procedures*⟩+≡
```
      subroutine cmd_plot_execute (plot, global)
        type(cmd_plot_t), intent(inout), target :: plot
        type(rt_data_t), intent(inout), target :: global
        real(default) :: lower_bound, upper_bound
        logical :: bounds_are_known
        type(string_t) :: title, description, xlabel, ylabel
        type(plot_labels_t) :: plot_labels
        call rt_data_link (plot%local, global)
        if (associated (plot%options)) then
           call command_list_execute (plot%options, plot%local)
```

```
      end if
      if (plot%use_id_expr) then
         call eval_tree_evaluate (plot%expr_id)
         plot%id = eval_tree_get_string (plot%expr_id)
      end if
      bounds_are_known = .true.
      call eval_tree_evaluate (plot%expr_lower_bound)
      call eval_tree_evaluate (plot%expr_upper_bound)
      if (eval_tree_result_is_known (plot%expr_lower_bound)) then
         lower_bound = eval_tree_get_real (plot%expr_lower_bound)
      else
         call msg_error ("Plot '" &
              // char (plot%id) // "': lower bound is undefined")
         bounds_are_known = .false.
      end if
      if (eval_tree_result_is_known (plot%expr_upper_bound)) then
         upper_bound = eval_tree_get_real (plot%expr_upper_bound)
      else
         call msg_error ("Plot '" &
              // char (plot%id) // "': upper bound is undefined")
         bounds_are_known = .false.
      end if
      title = var_list_get_sval (plot%local%var_list, var_str ("$title"))
      description = &
           var_list_get_sval (plot%local%var_list, var_str ("$description"))
      xlabel = var_list_get_sval (plot%local%var_list, var_str ("$xlabel"))
      ylabel = var_list_get_sval (plot%local%var_list, var_str ("$ylabel"))
      call plot_labels_init (plot_labels, title, &
           description, xlabel, ylabel)
      if (bounds_are_known) then
         call analysis_init_plot &
              (plot%id, lower_bound, upper_bound, plot_labels)
      else
         call msg_error ("Plot '" &
              // char (plot%id) // "': invalid declaration, skipping")
      end if
      call rt_data_restore (global, plot%local)
    end subroutine cmd_plot_execute
```

Analysis

Define the analysis expression. We store the parse tree for the right-hand side
instead of compiling it. Compilation is deferred to the process environment
where the analysis expression is used.

⟨*Commands: types*⟩+≡

```
    type :: cmd_analysis_t
       private
       type(parse_node_t), pointer :: pn_lexpr => null ()
    end type cmd_analysis_t
```

Output. Print the right-hand side as a parse tree.

⟨*Commands: procedures*⟩+≡

1239

```
subroutine cmd_analysis_write (analysis, unit, indent)
  type(cmd_analysis_t), intent(in) :: analysis
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "analysis ="
  call parse_node_write_rec (analysis%pn_lexpr, unit)
end subroutine cmd_analysis_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_analysis_compile (analysis, pn)
  type(cmd_analysis_t), pointer :: analysis
  type(parse_node_t), intent(in), target :: pn
  allocate (analysis)
  analysis%pn_lexpr => parse_node_get_sub_ptr (pn, 3)
end subroutine cmd_analysis_compile
```

Instead of evaluating the cut expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_analysis_execute (analysis, global)
  type(cmd_analysis_t), intent(inout), target :: analysis
  type(rt_data_t), intent(inout), target :: global
  global%pn_analysis_lexpr => analysis%pn_lexpr
end subroutine cmd_analysis_execute
```

Write analysis results

Depending on the argument, write one, several, or all analysis objects to a file. Write a LaTeX driver file as well.

⟨*Commands: types*⟩+≡
```
type :: cmd_write_analysis_t
  private
  integer :: n_args = 0
  type(string_t), dimension(:), allocatable :: id
  logical, dimension(:), allocatable :: use_id_expr
  type(eval_tree_t), dimension(:), allocatable :: expr_id
end type cmd_write_analysis_t
```

Finalizer for the eval trees.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_write_analysis_final (write)
  type(cmd_write_analysis_t), intent(inout) :: write
  integer :: i
  do i = 1, write%n_args
     call eval_tree_final (write%expr_id(i))
  end do
end subroutine cmd_write_analysis_final
```

Compile. Record the write ID and initialize lower, upper bound and bin width.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_write_analysis_compile (write, pn, global)
  type(cmd_write_analysis_t), pointer :: write
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(in), target :: global
  type(parse_node_t), pointer :: pn_args, pn_tag
  integer :: i
  pn_args => parse_node_get_sub_ptr (pn, 2)
  allocate (write)
  if (associated (pn_args)) then
     write%n_args = parse_node_get_n_sub (pn_args)
     allocate (write%id (write%n_args), write%use_id_expr (write%n_args))
     write%use_id_expr = .false.
     allocate (write%expr_id (write%n_args))
     pn_tag => parse_node_get_sub_ptr (pn_args)
     i = 1
     do while (associated (pn_tag))
        select case (char (parse_node_get_rule_key (pn_tag)))
        case ("analysis_id")
           write%id(i) = parse_node_get_string (pn_tag)
        case default
           write%use_id_expr(i) = .true.
           call eval_tree_init_sexpr &
                (write%expr_id(i), pn_tag, global%var_list)
        end select
        i = i + 1
        pn_tag => parse_node_get_next_ptr (pn_tag)
     end do
  end if
end subroutine cmd_write_analysis_compile
```

Command execution.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_write_analysis_execute (write, global)
  type(cmd_write_analysis_t), intent(inout), target :: write
  type(rt_data_t), intent(inout), target :: global
  type(string_t) :: filename, data_file, driver_file
  integer :: i, u_data, u_driver, u_log
  logical :: has_gmlcode
  u_log = logfile_unit ()
  if (var_list_is_known (global%var_list, &
                         var_str ("$analysis_filename"))) then
     filename = var_list_get_sval (global%var_list, &
                                   var_str ("$analysis_filename"))
     data_file = filename // ".dat"
     driver_file = filename // ".tex"
     call msg_message ("Writing analysis results to '" &
          // char (data_file) // "'")
     u_data = free_unit ()
     open (unit=u_data, file=char(data_file), &
           action="write", status="replace")
     call msg_message ("Writing analysis results display to '" &
```

```
              // char (driver_file) // "'")
      u_driver = free_unit ()
      open (unit=u_driver, file=char(driver_file), &
            action="write", status="replace")
   else
      u_data = -1
      u_driver = -1
   end if
   if (write%n_args == 0) then
      if (u_data >= 0) then
         call analysis_write (unit=u_data)
      else
         call analysis_write ()
         call analysis_write (unit=u_log)
         flush (u_log)
      end if
      if (u_driver >= 0) then
         call analysis_write_driver (data_file, unit=u_driver)
      end if
   else
      do i = 1, write%n_args
         if (write%use_id_expr(i)) then
            call eval_tree_evaluate (write%expr_id(i))
            write%id(i) = eval_tree_get_string (write%expr_id(i))
         end if
         if (u_data >= 0) then
            call analysis_write (write%id(i), unit=u_data)
         else
            call analysis_write (write%id(i))
            call analysis_write (write%id(i), unit=u_log)
            flush (u_log)
         end if
         if (u_driver >= 0) then
            call analysis_write_driver (data_file, write%id, unit=u_driver)
         end if
      end do
   end if
   if (u_data >= 0)  close (u_data)
   if (u_driver >= 0) then
      close (u_driver)
      if (write%n_args == 0) then
         has_gmlcode = analysis_has_plots ()
      else
         has_gmlcode = analysis_has_plots (write%id)
      end if
      call msg_message ("Compiling analysis results display in '" &
            // char (driver_file) // "'")
      call analysis_compile_tex (filename, has_gmlcode, global%os_data)
   end if
end subroutine cmd_write_analysis_execute
```

Clear analysis objects

The syntax is analogous to `write_analysis`. Each argument clears a particular
analysis object. No argument clears the whole analysis store. The objects are
cleared, but not deleted.

⟨*Commands: types*⟩+≡
```
type :: cmd_clear_t
   private
   integer :: n_args = 0
   type(string_t), dimension(:), allocatable :: id
   logical, dimension(:), allocatable :: use_id_expr
   type(eval_tree_t), dimension(:), allocatable :: expr_id
end type cmd_clear_t
```

Finalizer for the eval trees.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_clear_final (clear)
  type(cmd_clear_t), intent(inout) :: clear
  integer :: i
  do i = 1, clear%n_args
     call eval_tree_final (clear%expr_id(i))
  end do
end subroutine cmd_clear_final
```

Compile. Record the clear ID and initialize lower, upper bound and bin width.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_clear_compile (clear, pn, global)
  type(cmd_clear_t), pointer :: clear
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(in), target :: global
  type(parse_node_t), pointer :: pn_args, pn_tag
  type(string_t) :: key
  integer :: i
  pn_args => parse_node_get_sub_ptr (pn, 2)
  allocate (clear)
  if (associated (pn_args)) then
     clear%n_args = parse_node_get_n_sub (pn_args)
     allocate (clear%id (clear%n_args), clear%use_id_expr (clear%n_args))
     clear%use_id_expr = .false.
     allocate (clear%expr_id (clear%n_args))
     pn_tag => parse_node_get_sub_ptr (pn_args)
     i = 1
     do while (associated (pn_tag))
        key = parse_node_get_rule_key (pn_tag)
        select case (char (key))
        case ("iterations", "cuts", "weight", "scale", "analysis", "expect")
           clear%id(i) = key
        case ("analysis_id")
           clear%id(i) = parse_node_get_string (pn_tag)
        case default
           clear%use_id_expr(i) = .true.
           call eval_tree_init_sexpr &
                (clear%expr_id(i), pn_tag, global%var_list)
```

```
                 end select
                 i = i + 1
                 pn_tag => parse_node_get_next_ptr (pn_tag)
              end do
           end if
      end subroutine cmd_clear_compile
```

Command execution.
⟨*Commands: procedures*⟩+≡
```
      subroutine cmd_clear_execute (clear, global)
        type(cmd_clear_t), intent(inout), target :: clear
        type(rt_data_t), intent(inout), target :: global
        integer :: i
        if (clear%n_args == 0) then
           call analysis_clear ()
           call msg_message ("Cleared all analysis objects")
        else
           do i = 1, clear%n_args
              select case (char (clear%id(i)))
              case ("iterations")
                 call iterations_list_clear (global%it_list)
                 call msg_message ("Cleared iteration list setup")
              case ("cuts")
                 global%pn_cuts_lexpr => null ()
                 call msg_message ("Cleared cut setup")
              case ("weight")
                 global%pn_weight_expr => null ()
                 call msg_message ("Cleared integration weight setup")
              case ("scale")
                 global%pn_scale_expr => null ()
                 call msg_message ("Cleared event scale setup")
              case ("analysis")
                 global%pn_analysis_lexpr => null ()
                 call msg_message ("Cleared analysis setup")
              case ("expect")
                 call expect_clear ()
                 call msg_message ("Cleared counters of value checks")
              case default
                 if (clear%use_id_expr(i)) then
                    call eval_tree_evaluate (clear%expr_id(i))
                    clear%id(i) = eval_tree_get_string (clear%expr_id(i))
                 end if
                 call analysis_clear (clear%id(i))
                 call msg_message ("Cleared analysis object '" &
                      // char (clear%id(i)) // "'")
              end select
           end do
        end if
      end subroutine cmd_clear_execute
```

Record data

The expression syntax already contains a `record` keyword; this evaluates to a logical which is always true, but it has the side-effect of recording data into analysis objects. Here we define a command as an interface to this construct.

⟨*Commands: types*⟩+≡
```
    type :: cmd_record_t
       private
       type(eval_tree_t) :: lexpr
    end type cmd_record_t
```

Finalizer for the eval trees.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_record_final (record)
      type(cmd_record_t), intent(inout) :: record
      call eval_tree_final (record%lexpr)
    end subroutine cmd_record_final
```

Compile. Record the record ID and initialize lower, upper bound and bin width.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_record_compile (record, pn, global)
      type(cmd_record_t), pointer :: record
      type(parse_node_t), intent(in), target :: pn
      type(rt_data_t), intent(in), target :: global
      type(parse_node_t), pointer :: pn_lexpr, pn_lterm, pn_record
      call parse_node_create_branch (pn_lexpr, &
           syntax_get_rule_ptr (syntax_cmd_list, var_str ("lexpr")))
      call parse_node_create_branch (pn_lterm, &
           syntax_get_rule_ptr (syntax_cmd_list, var_str ("lterm")))
      call parse_node_append_sub (pn_lexpr, pn_lterm)
      pn_record => parse_node_get_sub_ptr (pn)
      call parse_node_append_sub (pn_lterm, pn_record)
      allocate (record)
      call eval_tree_init_lexpr (record%lexpr, pn_lexpr, global%var_list)
    end subroutine cmd_record_compile
```

Command execution.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_record_execute (record, global)
      type(cmd_record_t), intent(inout), target :: record
      type(rt_data_t), intent(inout), target :: global
      call eval_tree_evaluate (record%lexpr)
    end subroutine cmd_record_execute
```

Decay properties

This record holds decay properties for a particle. To be used for `unstable` and `stable` declarations, which contain an array of that type.

⟨*Commands: types*⟩+≡
```
    type :: decay_properties_t
```

```
      type(eval_tree_t) :: pdg
      type(string_t) :: prt
      type(flavor_t) :: flv
      integer :: n_proc = 0
      type(string_t), dimension(:), allocatable :: process_id
      real(default), dimension(:), allocatable :: br
    end type decay_properties_t
```

The finalizer is generic:

⟨*Commands: procedures*⟩+≡
```
    subroutine decay_properties_final (decay)
      type(decay_properties_t), intent(inout) :: decay
      call eval_tree_final (decay%pdg)
    end subroutine decay_properties_final
```

For completeness:

⟨*Commands: procedures*⟩+≡
```
    subroutine decay_properties_write (decay, unit, indent)
      type(decay_properties_t), intent(in) :: decay
      integer, intent(in), optional :: unit, indent
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      if (decay%n_proc /= 0) then
         write (u, "(1x,A)", advance="no") "unstable"
      else
         write (u, "(1x,A)", advance="no") "stable"
      end if
      write (u, "(1x,A)", advance="no")  char (decay%prt)
      if (decay%n_proc /= 0) then
         write (u, "(1x,A)", advance="no") "("
         do i = 1, decay%n_proc
            if (i /= 1)  write (u, "(', ')", advance="no")
            write (u, "(A)", advance="no")  char (decay%process_id(i))
         end do
         write (u, "(A)") ")"
      end if
    end subroutine decay_properties_write
```

Unstable particles

Mark a particle as unstable. For each unstable particle, we store a number of
decay channels and compute their respective BRs.

⟨*Commands: types*⟩+≡
```
    type :: cmd_unstable_t
      private
      type(decay_properties_t), dimension(:), allocatable :: decay
      type(command_list_t), pointer :: options => null ()
      type(rt_data_t) :: local
    end type cmd_unstable_t
```

Delete the eval tree.

⟨Commands: procedures⟩+≡

```
subroutine cmd_unstable_final (unstable)
  type(cmd_unstable_t), intent(inout) :: unstable
  integer :: d
  if (allocated (unstable%decay)) then
     do d = 1, size (unstable%decay)
        call decay_properties_final (unstable%decay(d))
     end do
     deallocate (unstable%decay)
  end if
  if (associated (unstable%options)) then
     call command_list_final (unstable%options)
     deallocate (unstable%options)
  end if
end subroutine cmd_unstable_final
```

Output.

⟨Commands: procedures⟩+≡

```
subroutine cmd_unstable_write (unstable, unit, indent)
  type(cmd_unstable_t), intent(in) :: unstable
  integer, intent(in), optional :: unit, indent
  integer :: d
  do d = 1, size (unstable%decay)
     call decay_properties_write (unstable%decay(d), unit, indent)
  end do
end subroutine cmd_unstable_write
```

Compile. Initiate an eval tree for the decaying particle and determine the decay channel process IDs.

⟨Commands: procedures⟩+≡

```
subroutine cmd_unstable_compile (unstable, pn, global)
  type(cmd_unstable_t), pointer :: unstable
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(in), target :: global
  type(parse_node_t), pointer :: pn_list, pn_decl
  type(parse_node_t), pointer :: pn_prt, pn_arg, pn_proc, pn_opt
  integer :: d, i
  pn_list => parse_node_get_sub_ptr (pn, 2)
  if (associated (pn_list)) then
     pn_opt => parse_node_get_next_ptr (pn_list)
  else
     pn_opt => null ()
  end if
  allocate (unstable)
  if (associated (pn_opt)) then
     allocate (unstable%options)
     call command_list_compile (unstable%options, pn_opt, unstable%local)
  end if
  call rt_data_local_init (unstable%local, global)
  allocate (unstable%decay (parse_node_get_n_sub (pn_list)))
  d = 0
  pn_decl => parse_node_get_sub_ptr (pn_list)
```

1247

```
      do while (associated (pn_decl))
         d = d + 1
         pn_prt => parse_node_get_sub_ptr (pn_decl)
         pn_arg => parse_node_get_next_ptr (pn_prt)
         call eval_tree_init_cexpr &
              (unstable%decay(d)%pdg, pn_prt, unstable%local%var_list)
         unstable%decay(d)%prt = "?"
         unstable%decay(d)%n_proc = parse_node_get_n_sub (pn_arg)
         allocate (unstable%decay(d)%process_id (unstable%decay(d)%n_proc))
         allocate (unstable%decay(d)%br (unstable%decay(d)%n_proc))
         pn_proc => parse_node_get_sub_ptr (pn_arg)
         do i = 1, unstable%decay(d)%n_proc
            unstable%decay(d)%process_id(i) = parse_node_get_string (pn_proc)
            pn_proc => parse_node_get_next_ptr (pn_proc)
         end do
         pn_decl => parse_node_get_next_ptr (pn_decl)
      end do
    end subroutine cmd_unstable_compile
```

Command execution. Evaluate the decaying particle and compute the BRs for
the decays, using previous integration runs. For each decay channel, initialize
event generation.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_unstable_execute (unstable, global)
      type(cmd_unstable_t), intent(inout), target :: unstable
      type(rt_data_t), intent(inout), target :: global
      type(pdg_array_t) :: aval
      type(flavor_t), dimension(:), allocatable :: flv_tmp
      type(flavor_t), dimension(:), allocatable :: flv
      real(default), dimension(:), allocatable :: integral
      real(default) :: integral_sum
      type(process_t), pointer :: process
      integer :: proc
      logical :: isotropic_decay, diagonal_decay
      type(particle_data_t), pointer :: prt_data
      type(decay_configuration_t), pointer :: decay_conf
      integer :: u, d
      u = logfile_unit ()
      call rt_data_link (unstable%local, global)
      if (associated (unstable%options)) then
         call command_list_execute (unstable%options, unstable%local)
      end if
      isotropic_decay = var_list_get_lval (unstable%local%var_list, &
           var_str ("?isotropic_decay"))
      diagonal_decay = var_list_get_lval (unstable%local%var_list, &
           var_str ("?diagonal_decay"))
      allocate (flv (size (unstable%decay)))
      do d = 1, size (unstable%decay)
         call eval_tree_evaluate (unstable%decay(d)%pdg)
         aval = eval_tree_get_pdg_array (unstable%decay(d)%pdg)
         call flavor_init (flv_tmp, aval, unstable%local%model)
         select case (size (flv_tmp))
         case (1);  flv(d) = flv_tmp(1)
```

```
      case default
         call pdg_array_write (aval)
         call msg_fatal ("Unstable particle expression " &
              // "does not evaluate to a unique particle")
         return
      end select
      prt_data => model_get_particle_ptr &
                       (unstable%local%model, flavor_get_pdg (flv(d)))
      if (associated (prt_data)) then
         if (flavor_is_antiparticle (flv(d))) then
            call particle_data_set (prt_data, &
                 a_is_stable = .false., &
                 a_decays_isotropically = isotropic_decay, &
                 a_decays_diagonal = diagonal_decay)
         else
            call particle_data_set (prt_data, &
                 p_is_stable = .false., &
                 p_decays_isotropically = isotropic_decay, &
                 p_decays_diagonal = diagonal_decay)
         end if
      else
         call msg_fatal ("Particle '" // char (unstable%decay(d)%prt) &
              // "' is not contained in model '" &
              // char (model_get_name (unstable%local%model)) // "'")
      end if
   end do
   do d = 1, size (unstable%decay)
      unstable%decay(d)%prt = flavor_get_name (flv(d))
      allocate (integral (unstable%decay(d)%n_proc))
      LOOP_PROC: do proc = 1, unstable%decay(d)%n_proc
         process => process_store_get_process_ptr &
                       (unstable%decay(d)%process_id(proc))
         if (associated (process)) then
            integral(proc) = process_get_integral (process)
            call process_setup_event_generation (process, qn_mask_in = &
                 new_quantum_numbers_mask (.false., .false., isotropic_decay, &
                                        mask_hd = diagonal_decay))
         else
            call msg_error ("Decay channel process '" // &
                 char (unstable%decay(d)%process_id(proc)) // "' is undefined")
            integral(proc) = 0
         end if
      end do LOOP_PROC
      integral_sum = sum (integral)
      if (integral_sum /= 0) then
         unstable%decay(d)%br = integral / integral_sum
      else
         call msg_fatal ("Unstable particle: Computed total width vanishes")
         unstable%decay(d)%br = 0
      end if
      call decay_store_append_decay &
           (flv(d), unstable%local%model, &
            integral_sum, unstable%decay(d)%n_proc, &
            isotropic_decay, diagonal_decay, &
```

```
            decay_conf)
        ASSIGN_DECAYS: do proc = 1, unstable%decay(d)%n_proc
           process => process_store_get_process_ptr &
                            (unstable%decay(d)%process_id(proc))
           if (associated (process)) then
              call decay_configuration_set_channel &
                    (decay_conf, proc, process, unstable%decay(d)%br(proc))
           end if
        end do ASSIGN_DECAYS
        deallocate (integral)
        call msg_message ("Particle '" // char (unstable%decay(d)%prt) &
              // "' declared as unstable.")
        call decay_configuration_write (decay_conf)
        call decay_configuration_write (decay_conf, u)
      end do
      call decay_store_recheck_final_state (verbose = .true.)
      call rt_data_restore (global, unstable%local)
    end subroutine cmd_unstable_execute
```

Stable particles

Mark a particle as stable. This is necessary only if it was previously marked as unstable.

⟨*Commands: types*⟩+≡
```
    type :: cmd_stable_t
      private
      type(decay_properties_t), dimension(:), allocatable :: decay
      type(command_list_t), pointer :: options => null ()
      type(rt_data_t) :: local
    end type cmd_stable_t
```

Delete the eval tree.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_stable_final (stable)
      type(cmd_stable_t), intent(inout) :: stable
      integer :: d
      if (allocated (stable%decay)) then
         do d = 1, size (stable%decay)
            call decay_properties_final (stable%decay(d))
         end do
         deallocate (stable%decay)
      end if
      if (associated (stable%options)) then
         call command_list_final (stable%options)
         deallocate (stable%options)
      end if
    end subroutine cmd_stable_final
```

Output.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_stable_write (stable, unit, indent)
```

```
    type(cmd_stable_t), intent(in) :: stable
    integer, intent(in), optional :: unit, indent
    integer :: d
    do d = 1, size (stable%decay)
        call decay_properties_write (stable%decay(d), unit, indent)
    end do
end subroutine cmd_stable_write
```

Compile. Initiate an eval tree for the particle code.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_stable_compile (stable, pn, global)
      type(cmd_stable_t), pointer :: stable
      type(parse_node_t), intent(in), target :: pn
      type(rt_data_t), intent(in), target :: global
      type(parse_node_t), pointer :: pn_list, pn_prt, pn_opt
      integer :: d
      pn_list => parse_node_get_sub_ptr (pn, 2)
      pn_opt => parse_node_get_next_ptr (pn_list)
      allocate (stable)
      call rt_data_local_init (stable%local, global)
      allocate (stable%decay (parse_node_get_n_sub (pn_list)))
      d = 0
      pn_prt => parse_node_get_sub_ptr (pn_list)
      do while (associated (pn_prt))
         d = d + 1
         call eval_tree_init_cexpr &
            (stable%decay(d)%pdg, pn_prt, stable%local%var_list)
         stable%decay(d)%prt = "?"
         pn_prt => parse_node_get_next_ptr (pn_prt)
      end do
    end subroutine cmd_stable_compile
```

Command execution. Evaluate the particle type and undo any previous unstable declaration.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_stable_execute (stable, global)
      type(cmd_stable_t), intent(inout), target :: stable
      type(rt_data_t), intent(inout), target :: global
      type(pdg_array_t) :: aval
      type(flavor_t), dimension(:), allocatable :: flv_tmp
      type(flavor_t) :: flv
      type(particle_data_t), pointer :: prt_data
      integer :: d
      call rt_data_link (stable%local, global)
      if (associated (stable%options)) then
         call command_list_execute (stable%options, stable%local)
      end if
      do d = 1, size (stable%decay)
         call eval_tree_evaluate (stable%decay(d)%pdg)
         aval = eval_tree_get_pdg_array (stable%decay(d)%pdg)
         call flavor_init (flv_tmp, aval, stable%local%model)
         select case (size (flv_tmp))
         case (1);  flv = flv_tmp(1)
```

```
          case default
             call pdg_array_write (aval)
             call msg_fatal ("Stable particle expression " &
                  // "does not evaluate to a unique particle")
             return
          end select
          stable%decay(d)%prt = flavor_get_name (flv)
          prt_data => &
               model_get_particle_ptr (stable%local%model, flavor_get_pdg (flv))
          if (associated (prt_data)) then
             if (flavor_is_antiparticle (flv)) then
                call particle_data_set (prt_data, a_is_stable=.true.)
                call particle_data_set (prt_data, a_decays_isotropically=.false.)
                call particle_data_set (prt_data, a_decays_diagonal=.false.)
             else
                call particle_data_set (prt_data, p_is_stable=.true.)
                call particle_data_set (prt_data, p_decays_isotropically=.false.)
                call particle_data_set (prt_data, p_decays_diagonal=.false.)
             end if
          else
             call msg_fatal ("Particle '" // char (stable%decay(d)%prt) &
                  // "' is not contained in model '" &
                  // char (model_get_name (stable%local%model)) // "'")
          end if
          call msg_message ("Particle '" // char (stable%decay(d)%prt) &
               // "' declared as stable.")
       end do
       call decay_store_recheck_final_state (verbose = .true.)
       call rt_data_restore (global, stable%local)
     end subroutine cmd_stable_execute
```

Parameters: formats for event-sample output

Specify all event formats that are to be used for output files in the subsequent simulation run. (The raw format is on by default and can be turned off here.)

⟨*Commands: types*⟩+≡
```
    type :: cmd_sample_format_t
      private
      type(string_t), dimension(:), allocatable :: format
      integer, dimension(:), allocatable :: fmt
    end type cmd_sample_format_t
```

Finalizer.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_sample_format_final (sample)
      type(cmd_sample_format_t), intent(inout) :: sample
      if (allocated (sample%format)) then
         deallocate (sample%format)
      end if
    end subroutine cmd_sample_format_final
```

Output.

⟨Commands: procedures⟩+≡

```
subroutine cmd_sample_format_write (sample, unit, indent)
  type(cmd_sample_format_t), intent(in) :: sample
  integer, intent(in), optional :: unit, indent
  integer :: u, i
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)", advance="no")  "sample ="
  if (allocated (sample%format)) then
     do i = 1, size (sample%format)
        write (u, "(1x,A)", advance="no")  char (sample%format(i))
     end do
     write (u, *)
  else
     write (u, "(1x,A)")  "[undefined]"
  end if
end subroutine cmd_sample_format_write
```

Compile. Initialize evaluation trees.

⟨Commands: procedures⟩+≡

```
subroutine cmd_sample_format_compile (sample, pn)
  type(cmd_sample_format_t), pointer :: sample
  type(parse_node_t), intent(in), target :: pn
  type(parse_node_t), pointer :: pn_arg
  type(parse_node_t), pointer :: pn_format
  integer :: i, n_format
  allocate (sample)
  pn_arg => parse_node_get_sub_ptr (pn, 3)
  if (associated (pn_arg)) then
     n_format = parse_node_get_n_sub (pn_arg)
     allocate (sample%format (n_format), sample%fmt (n_format))
     pn_format => parse_node_get_sub_ptr (pn_arg)
     i = 0
     do while (associated (pn_format))
        i = i + 1
        sample%format(i) = parse_node_get_string (pn_format)
        pn_format => parse_node_get_next_ptr (pn_format)
     end do
     sample%fmt = event_format_code (sample%format)
  else
     allocate (sample%format (0))
  end if
end subroutine cmd_sample_format_compile
```

Execute. Transform the string-valued format specifications into an integer array in the runtime data set.

⟨Commands: procedures⟩+≡

```
subroutine cmd_sample_format_execute (sample, global)
  type(cmd_sample_format_t), intent(inout) :: sample
  type(rt_data_t), intent(inout) :: global
  integer, dimension(:), allocatable :: fmt_tmp
  integer :: i, j, k
```

```
if (allocated (global%event_fmt))  deallocate (global%event_fmt)
allocate (fmt_tmp(size (sample%format)))
j = 0
TRANSFER: do i = 1, size (sample%format)
   if (sample%fmt(i) == FMT_NONE) then
      call msg_error ("Undefined event-file format '" &
           // char (sample%format(i)) // "' (skipped)")
      cycle TRANSFER
   end if
   if (j > 0) then
      do k = 1, j
         if (fmt_tmp(k) == sample%fmt(i)) then
            call msg_warning ("Ignoring multiple requests for " &
                 // "event-file format '" // char (sample%format(i)) &
                 // "'" )
            cycle TRANSFER
         end if
      end do
   end if
   j = j + 1
   fmt_tmp(j) = sample%fmt(i)
end do TRANSFER
if (j > 0) then
   allocate (global%event_fmt(j))
   global%event_fmt = fmt_tmp(1:j)
end if
deallocate (fmt_tmp)
end subroutine cmd_sample_format_execute
```

16.1.8   Simulation parameters

This transparent container holds the parameters that control event generation.

⟨*Commands: parameters*⟩+≡

```
integer, parameter :: NORM_UNDEFINED = 0
integer, parameter :: NORM_UNIT = 1
integer, parameter :: NORM_N_EVT = 2
integer, parameter :: NORM_SIGMA = 3
integer, parameter :: NORM_SIGMA_N_EVT = 4
```

⟨*Commands: types*⟩+≡

```
type :: simulation_parameters_t
  logical :: unweighted = .true.
  integer :: normalization_mode = NORM_UNDEFINED
  logical :: negative_weights = .false.
end type simulation_parameters_t
```

⟨*Commands: procedures*⟩+≡

```
recursive subroutine simulation_parameters_init &
    (sim, unweighted, event_normalization, negative_weights)
  type(simulation_parameters_t), intent(out) :: sim
  logical, intent(in) :: unweighted
  type(string_t), intent(in) :: event_normalization
```

```
      logical, intent(in) :: negative_weights
      sim%unweighted = unweighted
      select case (char (event_normalization))
      case ("auto", "Auto", "AUTO", "automatic", "Automatic", "AUTOMATIC")
         if (unweighted) then
            sim%normalization_mode = NORM_UNIT
         else
            sim%normalization_mode = NORM_SIGMA
         end if
      case ("1", "unity", "Unity", "UNITY")
         sim%normalization_mode = NORM_UNIT
      case ("1/n", "1/N")
         sim%normalization_mode = NORM_N_EVT
      case ("sigma", "Sigma", "SIGMA")
         sim%normalization_mode = NORM_SIGMA
      case ("sigma/n", "Sigma/n", "Sigma/N", "SIGMA/N")
         sim%normalization_mode = NORM_SIGMA_N_EVT
      case default
         call msg_error ("Unknown value '" // char (event_normalization) &
               // "for $event_normalization.  I'll assume 'auto'")
         call simulation_parameters_init &
               (sim, unweighted, var_str ("auto"), negative_weights)
      end select
    end subroutine simulation_parameters_init
```

⟨*Commands: procedures*⟩+≡

```
    subroutine simulation_parameters_write_message (sim, unit)
      type(simulation_parameters_t), intent(in) :: sim
      integer, intent(in), optional :: unit
      type(string_t) :: weight_str, norm_str, neg_str
      if (sim%unweighted) then
         weight_str = "unweighted"
      else
         weight_str = "weighted"
      end if
      select case (sim%normalization_mode)
      case (NORM_UNIT)
         norm_str = "1"
      case (NORM_N_EVT)
         norm_str = "1/n"
      case (NORM_SIGMA)
         norm_str = "sigma"
      case (NORM_SIGMA_N_EVT)
         norm_str = "sigma/n"
      case default
         norm_str = "unknown"
      end select
      if (sim%negative_weights) then
         neg_str = ", allow negative weights"
      else
         neg_str = ""
      end if
      call msg_message ("Simulation mode = " // char (weight_str) &
            // ", event_normalization = '" // char (norm_str) &
```

```
           // "'" // char (neg_str), &
           unit)
    end subroutine simulation_parameters_write_message
```

⟨*Commands: procedures*⟩+≡
```
    subroutine simulation_parameters_write (sim, unit)
      type(simulation_parameters_t), intent(in) :: sim
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, *) "Simulation parameters:"
      write (u, *) "  unweighted        = ", sim%unweighted
      write (u, *) "  normalization_mode = ", sim%normalization_mode
      write (u, *) "  negative_weights  = ", sim%negative_weights
    end subroutine simulation_parameters_write
```

⟨*Commands: procedures*⟩+≡
```
    function simulation_parameters_get_norm (sim, sigma, n) result (norm)
      real(default) :: norm
      type(simulation_parameters_t), intent(in) :: sim
      real(default), intent(in) :: sigma
      integer, intent(in) :: n
      select case (sim%normalization_mode)
      case (NORM_UNIT)
         norm = 1
      case (NORM_N_EVT)
         if (n /= 0) then
            norm = 1._default / n
         else
            norm = 1
         end if
      case (NORM_SIGMA)
         norm = sigma
      case (NORM_SIGMA_N_EVT)
         if (n /= 0) then
            norm = sigma / n
         else
            norm = sigma
         end if
      case default
         norm = 1
      end select
      if ((.not. sim%unweighted) .and. sigma /= 0)  norm = norm / sigma
    end function simulation_parameters_get_norm
```

⟨*Commands: procedures*⟩+≡
```
    function simulation_parameters_get_md5sum (sim) result (md5sum_sim)
      character(32) :: md5sum_sim
      type(simulation_parameters_t), intent(in) :: sim
      integer :: u
      u = free_unit ()
      open (u, status = "scratch")
      call simulation_parameters_write (sim, u)
```

```
       rewind (u)
       md5sum_sim = md5sum (u)
       close (u)
     end function simulation_parameters_get_md5sum
```

Simulation

⟨*Commands: types*⟩+≡
```
    type :: cmd_simulate_t
       private
       integer :: n_evt = 0
       integer :: n_proc = 0
       type(string_t), dimension(:), allocatable :: process_id
       type(command_list_t), pointer :: options => null ()
       type(rt_data_t) :: local
    end type cmd_simulate_t
```

Finalizer.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_simulate_final (simulate)
      type(cmd_simulate_t), intent(inout) :: simulate
      if (associated (simulate%options)) then
         call command_list_final (simulate%options)
         deallocate (simulate%options)
      end if
    end subroutine cmd_simulate_final
```

Output.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_simulate_write (simulate, unit, indent)
      type(cmd_simulate_t), intent(in) :: simulate
      integer, intent(in), optional :: unit, indent
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)", advance="no") "simulate ("
      do i = 1, simulate%n_proc
         if (i /= 1)  write (u, "(', ')", advance="no")
         write (u, "(A)", advance="no")  char (simulate%process_id(i))
      end do
      write (u, "(A)") ")"
      if (associated (simulate%options)) then
         write (u, "(1x,'{')")
         call command_list_write (simulate%options, unit, indent)
         call write_indent (u, indent)
         write (u, "(1x,'}')")
      end if
    end subroutine cmd_simulate_write
```

Compile. In contrast to WHIZARD 1 the confusing option to give the number
of unweighted events for weighted events as if unweighting were to take place has

1257

been abandoned. (We both use `n_events` for weighted and unweighted events, the variable `n_calls` from WHIZARD 1 has been discarded.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_simulate_compile (simulate, pn, global)
  type(cmd_simulate_t), pointer :: simulate
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(in), target :: global
  type(parse_node_t), pointer :: pn_proclist, pn_proc, pn_opt
  integer :: i
  pn_proclist => parse_node_get_sub_ptr (pn, 2)
  pn_opt => parse_node_get_next_ptr (pn_proclist)
  allocate (simulate)
  call rt_data_local_init (simulate%local, global)
  if (associated (pn_opt)) then
     allocate (simulate%options)
     call command_list_compile (simulate%options, pn_opt, simulate%local)
  end if
  simulate%n_proc = parse_node_get_n_sub (pn_proclist)
  allocate (simulate%process_id (simulate%n_proc))
  pn_proc => parse_node_get_sub_ptr (pn_proclist)
  do i = 1, simulate%n_proc
     simulate%process_id(i) = parse_node_get_string (pn_proc)
     pn_proc => parse_node_get_next_ptr (pn_proc)
  end do
end subroutine cmd_simulate_compile
```

Execute command: Simulate events.

TODO: support weighted events and negative weights.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_simulate_execute (simulate, global)
  type(cmd_simulate_t), intent(inout), target :: simulate
  type(rt_data_t), intent(inout), target :: global
  type(file_list_t) :: file_list
  type :: process_p
     type(process_t), pointer :: ptr
  end type process_p
  type(process_p), dimension(:), allocatable :: prc_array
  real(default), dimension(:), allocatable :: integral
  real(default) :: integral_sum, integral_cmp
  real(default) :: luminosity, lumi
  type(string_t) :: basename, file_raw, extension_raw
  type(decay_tree_t), dimension(:), allocatable, target :: decay_tree
  type(prt_list_t), target :: prt_list
  real(default), target :: event_weight
  type(eval_tree_t), target :: analysis_expr
  type(event_t), target :: event
  integer :: n_events, i_evt
  logical :: rebuild_events
  integer :: proc
  type(process_t), pointer :: process
  type(string_t) :: process_string
  integer :: n_in
  type(flavor_t), dimension(:), allocatable :: beam_flv
```

1258

```fortran
real(default), dimension(:), allocatable :: beam_energy
real(double) :: x
character(32), dimension(:), allocatable :: md5sum_process
character(32), dimension(:), allocatable :: md5sum_parameters
character(32), dimension(:), allocatable :: md5sum_results
character(32) :: md5sum_decays, md5sum_simulation
logical :: read_raw, write_raw, exist, ok
type(simulation_parameters_t) :: spar
integer :: i, u_raw, iostat
character(30) :: n_events_string
real(default) :: norm_weight
call rt_data_link (simulate%local, global)
simulate%local%n_processes = simulate%n_proc
if (associated (simulate%options)) then
   call command_list_execute (simulate%options, simulate%local)
end if
call simulation_parameters_init (spar, &
     var_list_get_lval (simulate%local%var_list, &
                        var_str ("?unweighted")), &
     var_list_get_sval (simulate%local%var_list, &
                        var_str ("$event_normalization")), &
     var_list_get_lval (simulate%local%var_list, &
                        var_str ("?negative_weights")))
do proc = 1, simulate%n_proc
   process => process_store_get_process_ptr (simulate%process_id(proc))
   if (.not. associated (process)) then
     call msg_fatal ("Process '" // char (simulate%process_id(proc)) &
          // "' must be integrated before simulation.")
     call rt_data_restore (global, simulate%local)
     return
   end if
   select case (proc)
   case (1)
      n_in = process_get_n_in (process)
      allocate (beam_flv (n_in), beam_energy (n_in))
      beam_flv = process_get_beam_flv (process)
      beam_energy = process_get_beam_energy (process)
   case default
      if (.not. process_has_matrix_element (process))  cycle
      if (process_get_n_in (process) /= n_in) then
         call msg_fatal ("Simulation: " &
              // "Mixture of scattering and decays")
         call rt_data_restore (global, simulate%local)
         return
      else if (any (process_get_beam_flv (process) /= beam_flv)) then
         call msg_fatal ("Simulation: " &
              // "Mismatch in beam particles")
         call rt_data_restore (global, simulate%local)
         return
      else if (any (process_get_beam_energy (process) /= beam_energy)) then
         call msg_fatal ("Simulation: " &
              // "Mismatch in beam energies")
         call rt_data_restore (global, simulate%local)
         return
```

```
      end if
   end select
end do

basename = var_list_get_sval &
     (simulate%local%var_list, var_str ("$sample"))
if (basename == "")  basename = simulate%process_id(1)
read_raw = var_list_get_lval &
     (simulate%local%var_list, var_str ("?read_raw")) &
     .and. .not. rebuild_events
write_raw = var_list_get_lval &
     (simulate%local%var_list, var_str ("?write_raw"))
extension_raw = var_list_get_sval &
     (simulate%local%var_list, var_str ("$extension_raw"))
file_raw = basename // "." // extension_raw
if (allocated (simulate%local%event_fmt)) then
   do i = 1, size (simulate%local%event_fmt)
      call file_list_append_file_spec (file_list, &
           simulate%process_id(1), &
           simulate%local%var_list, &
           simulate%local%event_fmt(i), &
           beam_flv, beam_energy, simulate%n_proc)
   end do
end if

allocate (prc_array (simulate%n_proc), integral (simulate%n_proc))
allocate (decay_tree (simulate%n_proc))
allocate (md5sum_process (simulate%n_proc))
allocate (md5sum_parameters (simulate%n_proc))
allocate (md5sum_results (simulate%n_proc))
process_string = ""
do proc = 1, simulate%n_proc
   if (proc > 1)  process_string = process_string // ", "
   process => process_store_get_process_ptr (simulate%process_id(proc))
   process_string = process_string // process_get_id (process)
end do
call msg_message ("Initializating simulation for processes " &
     // char (process_string) // ":")
call simulation_parameters_write_message (spar)
if (associated (simulate%local%pn_analysis_lexpr)) then
   call msg_message ("Using user-defined analysis setup.")
else
   call msg_message ("No analysis setup has been provided.")
end if
if (associated (simulate%local%pn_analysis_lexpr)) then
   call eval_tree_init_lexpr (analysis_expr, &
        simulate%local%pn_analysis_lexpr, simulate%local%var_list, &
        prt_list, event_weight)
end if
do proc = 1, simulate%n_proc
   process => process_store_get_process_ptr (simulate%process_id(proc))
   call process_setup_event_generation (process)
   call decay_tree_init (decay_tree(proc), process)
   prc_array(proc)%ptr => process
```

1260

```
      integral(proc) = process_get_integral (process)
      md5sum_process(proc) = process_get_md5sum (process)
      md5sum_parameters(proc) = process_get_md5sum_parameters (process)
      md5sum_results(proc) = process_get_md5sum_results (process)
   end do
   md5sum_decays = decay_store_get_md5sum ()
   md5sum_simulation = simulation_parameters_get_md5sum (spar)
   integral_sum = sum (integral)
   if (integral_sum <= 0) then
      call msg_error ("Event generation: " &
            // "sum of process integrals must be positive; skipping")
      return
   end if
   luminosity = var_list_get_rval &
         (simulate%local%var_list, var_str ("luminosity"))
   n_events = var_list_get_ival &
         (simulate%local%var_list, var_str ("n_events"))
   rebuild_events = var_list_get_lval &
         (simulate%local%var_list, var_str ("?rebuild_events"))
   simulate%n_evt = choose_n_evt (luminosity, n_events, integral_sum)
   if (integral_sum == 0) then
      lumi = 0
   else
      lumi = max(luminosity, n_events / integral_sum)
   end if

   call file_list_open (file_list, simulate)

   if (read_raw) then
      inquire (file = char (file_raw), exist = exist)
      if (exist) then
         call msg_message ("Reading events from file '" &
               // char (file_raw) // "' ...")
         u_raw = free_unit ()
         open (file = char (file_raw), unit = u_raw, form = "unformatted", &
                action = "read", status = "old")
         call raw_event_file_read_header (u_raw, &
               md5sum_process, md5sum_parameters, md5sum_results, &
               md5sum_decays, md5sum_simulation, &
               ok, iostat)
         if (iostat /= 0) then
            call msg_error ("Event file '" &
                  // char (file_raw) // "' is corrupt, discarding.")
            close (u_raw)
            read_raw = .false.
         else if (.not. ok) then
            close (u_raw)
            read_raw = .false.
         end if
      else
         read_raw = .false.
      end if
   end if
   i_evt = 0
```

```
if (read_raw) then
   READ_EVENTS: do while (i_evt < simulate%n_evt)
      call event_read_raw (event, u_raw, event_weight, iostat=iostat)
      if (iostat /= 0)  exit READ_EVENTS
      i_evt = i_evt + 1
      call event_do_analysis (event, prt_list, analysis_expr)
      call file_list_write_event (file_list, event, proc, i_evt)
      call event_final (event)
   end do READ_EVENTS
   write (msg_buffer, "(A,1x,I0,1x,A)")  "...", i_evt, "events read."
   call msg_message ()
end if

if (simulate%n_evt > i_evt) then
   write (n_events_string, "(I0)")  simulate%n_evt - i_evt
   call msg_message ("Generating " // trim (n_events_string) &
         // " events ...")
else
   write_raw = .false.
end if
if (write_raw) then
   call msg_message ("Writing events in internal format to file '" &
         // char (file_raw) // "'")
    write (msg_buffer, "(A,1x,G11.4)") &
               & "Event sample corresponds to luminosity [fb-1] = ", lumi
   call msg_message ()
   if (read_raw) then
      close (u_raw)
      open (file = char (file_raw), unit = u_raw, form = "unformatted", &
            action = "write", status = "old", position = "append")
   else
      u_raw = free_unit ()
      open (file = char (file_raw), unit = u_raw, form = "unformatted", &
            action = "write", status = "replace")
      call raw_event_file_write_header (u_raw, &
            md5sum_process, md5sum_parameters, md5sum_results, &
            md5sum_decays, md5sum_simulation)
   end if
else if (read_raw) then
   close (u_raw)
end if
do i = i_evt + 1, simulate%n_evt
   call tao_random_number (simulate%local%rng, x)
   integral_cmp = 0
   do proc = 1, simulate%n_proc
      integral_cmp = integral_cmp + integral(proc)
      if (integral_cmp > x * integral_sum)  exit
   end do
   if (proc > simulate%n_proc)  proc = simulate%n_proc
   process => prc_array(proc)%ptr
   call event_init (event, process, event_weight, decay_tree(proc))
   call event_generate &
           (event, simulate%local%rng, spar%unweighted, &
            FM_IGNORE_HELICITY, &
```

```
                  keep_correlations=.false., &
                  keep_virtual=.true.)
        norm_weight = &
             simulation_parameters_get_norm (spar, integral_sum, simulate%n_evt)
        event%weight = event%weight * norm_weight
        call event_do_analysis (event, prt_list, analysis_expr)
        call file_list_write_event (file_list, event, proc, i)
        if (write_raw)  call event_write_raw (event, u_raw)
        call event_final (event)
     end do

     if (write_raw)  close (u_raw)
     call file_list_close (file_list)
     do proc = 1, simulate%n_proc
        call decay_tree_final (decay_tree(proc))
     end do
     call eval_tree_final (analysis_expr)
     if (simulate%n_evt > i_evt)  call msg_message ("... done")
     call msg_message ("Simulation finished.")
     call rt_data_restore (global, simulate%local)
   end subroutine cmd_simulate_execute
```

Choose the number of events to generate from either the luminosity or the specified **n_events**, whatever is larger.

⟨*Commands: procedures*⟩+≡
```
   function choose_n_evt (luminosity, n_evt_in, integral) result (n_evt)
     integer :: n_evt
     real(default), intent(in) :: luminosity
     integer, intent(in) :: n_evt_in
     real(default), intent(in) :: integral
     n_evt = max (nint (luminosity * integral), n_evt_in)
   end function choose_n_evt
```

Parameters: random-number generator seed

Specify a new random-number generator seed and set it.

⟨*Commands: types*⟩+≡
```
   type :: cmd_seed_t
      private
      type(eval_tree_t) :: expr
   end type cmd_seed_t
```

Finalizer.

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_seed_final (seed)
     type(cmd_seed_t), intent(inout) :: seed
     call eval_tree_final (seed%expr)
   end subroutine cmd_seed_final
```

Output.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_seed_write (seed, unit, indent)
  type(cmd_seed_t), intent(in) :: seed
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "seed ="
  call eval_tree_write (seed%expr, unit)
end subroutine cmd_seed_write
```

Compile. Initialize evaluation trees.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_seed_compile (seed, pn, global)
  type(cmd_seed_t), pointer :: seed
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(in), target :: global
  type(parse_node_t), pointer :: pn_expr
  allocate (seed)
  pn_expr => parse_node_get_sub_ptr (pn, 3)
  call eval_tree_init_expr (seed%expr, pn_expr, global%var_list)
end subroutine cmd_seed_compile
```

Execute. Evaluate the trees and add the result to the iteration list in the runtime
data set.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_seed_execute (seed, global)
  type(cmd_seed_t), intent(inout) :: seed
  type(rt_data_t), intent(inout) :: global
  integer :: seed_val
  call eval_tree_evaluate (seed%expr)
  if (eval_tree_result_is_known (seed%expr)) then
     seed_val = eval_tree_get_int (seed%expr)
     call set_rng_seed (global%rng, global%var_list, seed_val, verbose=.true.)
  else
     call msg_error ("Setting seed value: " &
          // "undefined result of integer expression evaluation")
  end if
end subroutine cmd_seed_execute
```

⟨*Commands: procedures*⟩+≡
```
subroutine set_rng_seed (rng, var_list, seed_val, verbose)
  type(tao_random_state), intent(inout) :: rng
  type(var_list_t), intent(inout), target :: var_list
  integer, intent(in) :: seed_val
  logical, intent(in) :: verbose
  character(30) :: buffer
  if (verbose) then
     write (buffer, "(I0)")  seed_val
     call msg_message ("Setting seed for random-number generator to " &
          // trim (buffer))
```

```
      end if
      call tao_random_seed (rng, seed_val)
      call var_list_set_int (var_list, var_str ("seed_value"), &
           seed_val, is_known=.true.)
   end subroutine set_rng_seed
```

Parameters: number of iterations

Specify number of iterations and number of calls for one integration pass.

⟨*Commands: types*⟩+≡
```
   type :: cmd_iterations_t
     private
     integer :: n_pass = 0
     type(eval_tree_t), dimension(:), allocatable :: expr_n_it
     type(eval_tree_t), dimension(:), allocatable :: expr_n_calls
   end type cmd_iterations_t
```

Finalizer.

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_iterations_final (iterations)
     type(cmd_iterations_t), intent(inout) :: iterations
     integer :: i
     if (allocated (iterations%expr_n_it)) then
        do i = 1, size (iterations%expr_n_it)
           call eval_tree_final (iterations%expr_n_it(i))
        end do
     end if
     if (allocated (iterations%expr_n_calls)) then
        do i = 1, size (iterations%expr_n_calls)
           call eval_tree_final (iterations%expr_n_calls(i))
        end do
     end if
   end subroutine cmd_iterations_final
```

Output.

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_iterations_write (iterations, unit, indent)
     type(cmd_iterations_t), intent(in) :: iterations
     integer, intent(in), optional :: unit, indent
     integer :: u, i
     u = output_unit (unit);  if (u < 0)  return
     call write_indent (u, indent)
     write (u, "(1x,A)")  "iterations ="
     do i = 1, iterations%n_pass
        call eval_tree_write (iterations%expr_n_it(i), unit)
        write (u, "(1x,A)") ":"
        call write_indent (u, indent)
        call eval_tree_write (iterations%expr_n_calls(i), unit)
        call write_indent (u, indent)
        if (i < iterations%n_pass)  write (u, "(1x,A)") ":"
     end do
```

```
        write (u, "(1x,A)") ")"
    end subroutine cmd_iterations_write
```

Compile. Initialize evaluation trees.

⟨Commands: procedures⟩+≡
```
    subroutine cmd_iterations_compile (iterations, pn, global)
      type(cmd_iterations_t), pointer :: iterations
      type(parse_node_t), intent(in), target :: pn
      type(rt_data_t), intent(in), target :: global
      type(parse_node_t), pointer :: pn_arg, pn_n_it, pn_n_calls
      type(parse_node_t), pointer :: pn_it_spec, pn_calls_spec
      integer :: i
      allocate (iterations)
      pn_arg => parse_node_get_sub_ptr (pn, 3)
      if (associated (pn_arg)) then
         iterations%n_pass = parse_node_get_n_sub (pn_arg)
         allocate (iterations%expr_n_it (iterations%n_pass))
         allocate (iterations%expr_n_calls (iterations%n_pass))
         pn_it_spec => parse_node_get_sub_ptr (pn_arg)
         i = 1
         do while (associated (pn_it_spec))
            select case (char (parse_node_get_rule_key (pn_it_spec)))
            case ("it_spec")
               pn_n_it => parse_node_get_sub_ptr (pn_it_spec)
               pn_calls_spec => parse_node_get_next_ptr (pn_n_it)
            case ("calls_spec")
               pn_n_it => null ()
               pn_calls_spec => pn_it_spec
            end select
            if (associated (pn_calls_spec)) then
               pn_n_calls => parse_node_get_sub_ptr (pn_calls_spec, 2)
            else
               pn_n_calls => null ()
            end if
            if (associated (pn_n_it)) then
               call eval_tree_init_expr (iterations%expr_n_it(i), &
                    pn_n_it, global%var_list)
            end if
            if (associated (pn_n_calls)) then
               call eval_tree_init_expr (iterations%expr_n_calls(i), &
                    pn_n_calls, global%var_list)
            end if
            i = i + 1
            pn_it_spec => parse_node_get_next_ptr (pn_it_spec)
         end do
      else
         allocate (iterations%expr_n_it (0), iterations%expr_n_calls (0))
      end if
    end subroutine cmd_iterations_compile
```

Execute. Evaluate the trees and add the result to the iteration list in the runtime data set.

⟨Commands: procedures⟩+≡

```
subroutine cmd_iterations_execute (iterations, global)
  type(cmd_iterations_t), intent(inout) :: iterations
  type(rt_data_t), intent(inout) :: global
  integer, dimension(iterations%n_pass) :: n_it, n_calls
  integer :: i
  do i = 1, iterations%n_pass
     call eval_tree_evaluate (iterations%expr_n_it(i))
     call eval_tree_evaluate (iterations%expr_n_calls(i))
     if (eval_tree_result_is_known (iterations%expr_n_it(i))) then
        n_it(i) = eval_tree_get_int (iterations%expr_n_it(i))
     else
        n_it(i) = 0
     end if
     if (eval_tree_result_is_known (iterations%expr_n_calls(i))) then
        n_calls(i) = eval_tree_get_int (iterations%expr_n_calls(i))
     else
        n_calls(i) = 0
     end if
  end do
  call iterations_list_init (global%it_list, n_it, n_calls)
end subroutine cmd_iterations_execute
```

Scan over parameters and other objects

⟨*Commands: parameters*⟩+≡
```
integer, parameter :: STEP_NONE = 0
integer, parameter :: STEP_ADD = 1
integer, parameter :: STEP_SUB = 2
integer, parameter :: STEP_MUL = 3
integer, parameter :: STEP_DIV = 4
```

⟨*Commands: types*⟩+≡
```
type :: cmd_scan_t
   private
   integer :: var_type = V_NONE
   type(string_t) :: var_name
   logical :: allow_steps = .false.
   integer :: n_arg = 0
   type(command_list_t), dimension(:), pointer :: cmd_var => null ()
   logical, dimension(:), allocatable :: has_range
   type(eval_tree_t), dimension(:), allocatable :: beg_expr
   type(eval_tree_t), dimension(:), allocatable :: end_expr
   integer, dimension(:), allocatable :: step_type
   type(eval_tree_t), dimension(:), allocatable :: step_expr
   type(command_list_t), pointer :: body => null ()
   type(rt_data_t) :: local
end type cmd_scan_t
```

Finalizer.

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_scan_final (loop)
  type(cmd_scan_t), intent(inout) :: loop
  integer :: i
```

```
        if (associated (loop%cmd_var)) then
           do i = 1, size (loop%cmd_var)
              call command_list_final (loop%cmd_var(i))
           end do
           deallocate (loop%cmd_var)
        end if
        if (allocated (loop%has_range))  deallocate (loop%has_range)
        if (allocated (loop%beg_expr)) then
           do i = 1, size (loop%beg_expr)
              call eval_tree_final (loop%beg_expr(i))
           end do
           deallocate (loop%beg_expr)
        end if
        if (allocated (loop%end_expr)) then
           do i = 1, size (loop%end_expr)
              call eval_tree_final (loop%end_expr(i))
           end do
           deallocate (loop%end_expr)
        end if
        if (allocated (loop%step_type))  deallocate (loop%step_type)
        if (allocated (loop%step_expr)) then
           do i = 1, size (loop%step_expr)
              call eval_tree_final (loop%step_expr(i))
           end do
           deallocate (loop%step_expr)
        end if
        if (associated (loop%body)) then
           call command_list_final (loop%body)
           deallocate (loop%body)
        end if
     end subroutine cmd_scan_final
```

Output.
⟨*Commands: procedures*⟩+≡
```
     subroutine cmd_scan_write (loop, unit, indent)
       type(cmd_scan_t), intent(in) :: loop
       integer, intent(in), optional :: unit, indent
       integer :: u
       u = output_unit (unit);  if (u < 0)  return
       call write_indent (u, indent)
       write (u, "(A)", advance="no") "scan ("
       if (associated (loop%body)) then
          write (u, "(1x,'{')")
          call command_list_write (loop%body, unit, indent)
          call write_indent (u, indent)
          write (u, "(1x,'}')")
       else
          write (u, *)
       end if
     end subroutine cmd_scan_write
```

Compile the loop, including the list of step specifications.
⟨*Commands: procedures*⟩+≡

```fortran
recursive subroutine cmd_scan_compile (loop, pn, global)
  type(cmd_scan_t), pointer :: loop
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_spec, pn_cmd, pn_list, pn_body
  integer :: i
  type(parse_tree_t) :: parse_tree
  type(parse_node_t), pointer :: pn_root, pn_decl, pn_init_arg, pn_arg
  type(parse_node_t), pointer :: pn_expr, pn_range, pn_range_expr
  type(parse_node_t), pointer :: pn_step, pn_step_op, pn_step_expr
  type(parse_node_t), pointer :: pn_var, pn_name
  type(string_t) :: key, str_init
  type(lexer_t) :: lexer
  type(stream_t), target :: stream
  logical :: declaration, new
  pn_spec => parse_node_get_sub_ptr (pn)
  pn_cmd => parse_node_get_sub_ptr (pn_spec, 2)
  pn_var => pn_cmd
  allocate (loop)
  call rt_data_local_init (loop%local, global)
  if (associated (pn_cmd)) then
     key = parse_node_get_rule_key (pn_cmd)
  else
     loop%var_type = V_NONE
     loop%n_arg = 0
     return
  end if
  declaration = .false.
  select case (char (key))
  case ("cmd_model_list")
     str_init = 'model = ""'
  case ("cmd_library_list")
     str_init = 'library = ""'
  case ("cmd_seed_list")
     loop%var_type = V_INT
     loop%var_name = "seed"
     str_init = 'seed = 0'
     loop%allow_steps = .true.
  case ("cmd_cuts_list")
     str_init = 'cuts = true'
  case ("cmd_weight_list")
     str_init = 'weight = 0'
  case ("cmd_scale_list")
     str_init = 'scale = 0'
  case ("cmd_analysis_list")
     str_init = 'analysis = true'
  case default
     new = .false.
     select case (char (key))
     case ("log_decl_list")
        loop%var_type = V_LOG
        pn_var => parse_node_get_sub_ptr (pn_cmd, 2)
        pn_name => parse_node_get_sub_ptr (pn_var, 2)
        loop%var_name = '?' // parse_node_get_string (pn_name)
```

1269

```
         str_init = 'logical ' // loop%var_name // ' = true'
         declaration = .true.
         new = .true.
      case ("log_list")
         loop%var_type = V_LOG
         pn_name => parse_node_get_sub_ptr (pn_cmd, 2)
         loop%var_name = '?' // parse_node_get_string (pn_name)
         str_init = loop%var_name // ' = true'
      case ("int_list")
         loop%var_type = V_INT
         pn_name => parse_node_get_sub_ptr (pn_cmd, 2)
         loop%var_name = parse_node_get_string (pn_name)
         str_init = 'int ' // loop%var_name // ' = 0'
         loop%allow_steps = .true.
         new = .true.
      case ("real_list")
         loop%var_type = V_REAL
         pn_name => parse_node_get_sub_ptr (pn_cmd, 2)
         loop%var_name = parse_node_get_string (pn_name)
         str_init = 'real ' // loop%var_name // ' = 0'
         loop%allow_steps = .true.
         new = .true.
      case ("num_list")
         loop%var_type = V_REAL
         pn_name => parse_node_get_sub_ptr (pn_cmd)
         loop%var_name = parse_node_get_string (pn_name)
         str_init = loop%var_name // ' = 0'
         loop%allow_steps = .true.
      case ("string_decl_list")
         loop%var_type = V_STR
         pn_var => parse_node_get_sub_ptr (pn_cmd, 2)
         pn_name => parse_node_get_sub_ptr (pn_var, 2)
         loop%var_name = '$' // parse_node_get_string (pn_name)
         str_init = 'string ' // loop%var_name // ' = ""'
         declaration = .true.
         new = .true.
      case ("string_list")
         loop%var_type = V_STR
         pn_name => parse_node_get_sub_ptr (pn_cmd, 2)
         loop%var_name = '$' // parse_node_get_string (pn_name)
         str_init = loop%var_name // ' = ""'
      case ("alias_list")
         loop%var_type = V_PDG
         pn_name => parse_node_get_sub_ptr (pn_cmd, 2)
         loop%var_name = parse_node_get_string (pn_name)
         str_init = 'alias ' // loop%var_name // ' = PDG(0)'
         new = .true.
      case default
         call parse_node_mismatch &
              ("scan: model|library|seed|cuts|weight|scale|analysis|" &
               // "variable",  pn_cmd)
      end select
      call var_list_check_user_var &
           (global%var_list, loop%var_name, loop%var_type, new)
```

```
        if (loop%var_type == V_NONE) then
            call msg_fatal ("Invalid scan variable declaration")
            loop%n_arg = 0
            return
        end if
    end select
    if (associated (pn_var)) then
        pn_list => parse_node_get_last_sub_ptr (pn_var)
        if (associated (pn_list)) then
            loop%n_arg = parse_node_get_n_sub (pn_list)
            allocate (loop%cmd_var (loop%n_arg))
            call lexer_init_cmd_list (lexer)
            allocate (loop%has_range (loop%n_arg)); loop%has_range = .false.
            if (loop%allow_steps) then
                allocate (loop%beg_expr (loop%n_arg))
                allocate (loop%end_expr (loop%n_arg))
                allocate (loop%step_type (loop%n_arg)); loop%step_type = STEP_NONE
                allocate (loop%step_expr (loop%n_arg))
            end if
            call stream_init (stream, str_init)
            call lexer_assign_stream (lexer, stream)
            call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
            pn_root => parse_tree_get_root_ptr (parse_tree)
            if (declaration) then
                pn_decl => parse_node_get_sub_ptr (pn_root)
                pn_cmd => parse_node_get_sub_ptr (pn_decl, 2)
            else
                pn_cmd => parse_node_get_sub_ptr (pn_root)
            end if
            pn_init_arg => parse_node_get_last_sub_ptr (pn_cmd)
            i = 0
            pn_arg => parse_node_get_sub_ptr (pn_list)
            do while (associated (pn_arg))
                i = i + 1
                select case (char (parse_node_get_rule_key (pn_arg)))
                case ("num_steps")
                    pn_expr => parse_node_get_sub_ptr (pn_arg)
                    pn_range => parse_node_get_next_ptr (pn_expr)
                    if (associated (pn_range)) then
                        loop%has_range(i) = .true.
                        pn_range_expr => parse_node_get_sub_ptr (pn_range, 2)
                        call eval_tree_init_expr (loop%beg_expr(i), &
                             pn_expr, loop%local%var_list)
                        call eval_tree_init_expr (loop%end_expr(i), &
                             pn_range_expr, loop%local%var_list)
                        pn_step => parse_node_get_next_ptr (pn_range_expr)
                        if (associated (pn_step)) then
                            pn_step_op => parse_node_get_sub_ptr (pn_step)
                            select case (char (parse_node_get_key (pn_step_op)))
                            case ("/+");  loop%step_type(i) = STEP_ADD
                            case ("/-");  loop%step_type(i) = STEP_SUB
                            case ("/*");  loop%step_type(i) = STEP_MUL
                            case ("//");  loop%step_type(i) = STEP_DIV
                            end select
```

```
                          pn_step_expr => parse_node_get_next_ptr (pn_step_op)
                          call eval_tree_init_expr (loop%step_expr(i), &
                               pn_step_expr, loop%local%var_list)
                     end if
                  end if
               case default
                  pn_expr => pn_arg
               end select
               call parse_node_replace_last_sub (pn_cmd, pn_expr)
               call command_list_compile (loop%cmd_var(i), pn_root, loop%local)
               pn_arg => parse_node_get_next_ptr (pn_arg)
            end do
            call parse_node_replace_last_sub (pn_cmd, pn_init_arg)
            call parse_tree_final (parse_tree)
            call stream_final (stream)
            call lexer_final (lexer)
         end if
         pn_body => parse_node_get_next_ptr (pn_spec)
         if (associated (pn_body)) then
            allocate (loop%body)
            call command_list_compile (loop%body, pn_body, loop%local)
         end if
      else
         loop%n_arg = 0
      end if
    end subroutine cmd_scan_compile
```

Execute the loop for all values in the step list.

⟨*Commands: procedures*⟩+≡

```
    recursive subroutine cmd_scan_execute (loop, global)
      type(cmd_scan_t), intent(inout), target :: loop
      type(rt_data_t), intent(inout), target :: global
      type(string_t) :: model_name
      logical :: is_seed
      integer :: i, j
      integer :: i1, i2, istep, ival
      real(default) :: r1, r2, rstep, rval
      type(var_entry_t), pointer :: var
      type(var_list_t), pointer :: model_vars
      call rt_data_link (loop%local, global)
      if (associated (loop%local%model)) then
         model_name = model_get_name (loop%local%model)
         model_vars => model_get_var_list_ptr (loop%local%model)
      else
         model_vars => null ()
      end if
      LOOP_LIST: do i = 1, loop%n_arg
         call command_list_execute (loop%cmd_var(i), loop%local)
         if (.not. loop%has_range(i)) then
            if (associated (loop%body)) then
               call command_list_execute (loop%body, loop%local)
               if (loop%local%quit) then
                  global%quit_code = loop%local%quit_code
                  global%quit = .true.
```

```
            return
         end if
      end if
else
   call eval_tree_evaluate (loop%beg_expr(i))
   if (eval_tree_result_is_known (loop%beg_expr(i))) then
      r1 = eval_tree_get_real (loop%beg_expr(i))
      i1 = eval_tree_get_int  (loop%beg_expr(i))
   else
      call msg_error ("Scan: undefined lower bound, skipping")
      cycle LOOP_LIST
   end if
   call eval_tree_evaluate (loop%end_expr(i))
   if (eval_tree_result_is_known (loop%end_expr(i))) then
      r2 = eval_tree_get_real (loop%end_expr(i))
      i2 = eval_tree_get_int  (loop%end_expr(i))
   else
      call msg_error ("Scan: undefined upper bound, skipping")
      cycle LOOP_LIST
   end if
   select case (loop%step_type(i))
   case (STEP_NONE)
      rstep = 1
      istep = 1
   case default
      call eval_tree_evaluate (loop%step_expr(i))
      if (eval_tree_result_is_known (loop%step_expr(i))) then
         rstep = eval_tree_get_real (loop%step_expr(i))
         istep = eval_tree_get_int  (loop%step_expr(i))
      else
         call msg_error ("Scan: undefined step size, skipping")
         cycle LOOP_LIST
      end if
   end select
   if (bounds_check_fails (loop%step_type(i), loop%var_type)) &
        cycle LOOP_LIST
   if (loop%var_name == "seed") then
      is_seed = .true.
   else
      is_seed = .false.
      var => var_list_get_var_ptr (loop%local%var_list, loop%var_name)
      if (var_entry_get_type (var) /= loop%var_type) then
         call msg_fatal ("Type mismatch for loop variable '" &
              // char (loop%var_name) // "'; skipping")
         exit LOOP_LIST
      end if
   end if
   select case (loop%var_type)
   case (V_REAL)
      select case (loop%step_type(i))
      case (STEP_NONE, STEP_ADD)
         do j = 0, huge (0)
            rval = r1 + j * rstep
            if (rstep > 0) then
```

```
            if (rval > r2) exit
         else
            if (rval < r2) exit
         end if
         call set_real (rval, j/=0)
         if (associated (loop%body)) then
            call command_list_execute (loop%body, loop%local)
            if (loop%local%quit) then
               global%quit_code = loop%local%quit_code
               global%quit = .true.
               return
            end if
         end if
      end do
case (STEP_SUB)
   do j = 0, huge (0)
      rval = r1 - j * rstep
      if (rstep > 0) then
         if (rval < r2) exit
      else
         if (rval > r2) exit
      end if
      call set_real (rval, j/=0)
      if (associated (loop%body)) then
         call command_list_execute (loop%body, loop%local)
         if (loop%local%quit) then
            global%quit_code = loop%local%quit_code
            global%quit = .true.
            return
         end if
      end if
   end do
case (STEP_MUL)
   rval = r1
   do j = 0, huge (0)
      if (rstep > 1) then
         if (rval > r2)  exit
      else
         if (rval < r2) exit
      end if
      call set_real (rval, j/=0)
      if (associated (loop%body)) then
         call command_list_execute (loop%body, loop%local)
         if (loop%local%quit) then
            global%quit_code = loop%local%quit_code
            global%quit = .true.
            return
         end if
      end if
      rval = rval * rstep
   end do
case (STEP_DIV)
   rval = r1
   do j = 0, huge (0)
```

```
             if (rstep > 1) then
                if (rval < r2)  exit
             else
                if (rval > r2) exit
             end if
             call set_real (rval, j/=0)
             if (associated (loop%body)) then
                call command_list_execute (loop%body, loop%local)
                if (loop%local%quit) then
                   global%quit_code = loop%local%quit_code
                   global%quit = .true.
                   return
                end if
             end if
             rval = rval / rstep
          end do
       end select
    case (V_INT)
       select case (loop%step_type(i))
       case (STEP_NONE, STEP_ADD)
          do j = 0, huge (0)
             ival = i1 + j * istep
             if (istep > 0) then
                if (ival > i2) exit
             else
                if (ival < i2) exit
             end if
             call set_int (ival, j/=0, is_seed)
             if (associated (loop%body)) then
                call command_list_execute (loop%body, loop%local)
                if (loop%local%quit) then
                   global%quit_code = loop%local%quit_code
                   global%quit = .true.
                   return
                end if
             end if
          end do
       case (STEP_SUB)
          do j = 0, huge (0)
             ival = i1 - j * istep
             if (istep > 0) then
                if (ival < i2) exit
             else
                if (ival > i2) exit
             end if
             call set_int (ival, j/=0, is_seed)
             if (associated (loop%body)) then
                call command_list_execute (loop%body, loop%local)
                if (loop%local%quit) then
                   global%quit_code = loop%local%quit_code
                   global%quit = .true.
                   return
                end if
             end if
```

```
                  end do
              case (STEP_MUL)
                  ival = i1
                  do j = 0, huge (0)
                      if (istep > 1) then
                          if (ival > i2)  exit
                      else
                          if (ival < i2) exit
                      end if
                      call set_int (ival, j/=0, is_seed)
                      if (associated (loop%body)) then
                          call command_list_execute (loop%body, loop%local)
                          if (loop%local%quit) then
                              global%quit_code = loop%local%quit_code
                              global%quit = .true.
                              return
                          end if
                      end if
                      ival = ival * istep
                  end do
              case (STEP_DIV)
                  ival = i1
                  do j = 0, huge (0)
                      if (istep > 1) then
                          if (ival < i2)  exit
                      else
                          if (ival > i2) exit
                      end if
                      call set_int (ival, j/=0, is_seed)
                      if (associated (loop%body)) then
                          call command_list_execute (loop%body, loop%local)
                          if (loop%local%quit) then
                              global%quit_code = loop%local%quit_code
                              global%quit = .true.
                              return
                          end if
                      end if
                      ival = ival / istep
                  end do
              end select
          end select
      end if
  end do LOOP_LIST
  call rt_data_restore (global, loop%local)
contains
  subroutine set_real (rval, verbose)
    real(default), intent(in) :: rval
    logical, intent(in) :: verbose
    call var_entry_set_real (var, rval, is_known=.true., verbose=verbose, &
        model_name=model_name)
    if (var_entry_is_copy (var)) then
        call model_parameters_update (loop%local%model)
        call var_list_synchronize (loop%local%var_list, model_vars)
    end if
```

```fortran
      end subroutine set_real
    subroutine set_int (ival, verbose, is_seed)
      integer, intent(in) :: ival
      logical, intent(in) :: verbose, is_seed
      if (is_seed) then
         call set_rng_seed (loop%local%rng, loop%local%var_list, ival, &
              verbose=verbose)
      else
         call var_entry_set_int (var, ival, is_known=.true., verbose=verbose, &
              model_name=model_name)
         if (var_entry_is_copy (var)) then
            call model_parameters_update (loop%local%model)
            call var_list_synchronize (loop%local%var_list, model_vars)
         end if
      end if
    end subroutine set_int
    function bounds_check_fails (step_type, var_type) result (fails)
      logical :: fails
      integer, intent(in) :: step_type, var_type
      fails = .false.
      select case (var_type)
      case (V_REAL)
         if (rstep == 0) then
            call msg_error ("Scan: step size equals zero, skipping")
            fails = .true.;  return
         end if
         select case (step_type)
         case (STEP_MUL, STEP_DIV)
            if (rstep < 0) then
               call msg_error ("Scan: multiplicative step < 0, skipping")
               fails = .true.;  return
            else if (rstep == 1) then
               call msg_error ("Scan: multiplicative step = 1, skipping")
               fails = .true.;  return
            else if (r1 == 0 .or. r2 == 0) then
               call msg_error ("Scan: " &
                    // "boundary for multiplicative step is zero, skipping")
               fails = .true.;  return
            end if
         end select
      case (V_INT)
         if (istep == 0) then
            call msg_error ("Scan: step size equals zero, skipping")
            fails = .true.;  return
         end if
         select case (step_type)
         case (STEP_MUL, STEP_DIV)
            if (istep < 0) then
               call msg_error ("Scan: multiplicative step < 0, skipping")
               fails = .true.;  return
            else if (istep == 1) then
               call msg_error ("Scan: multiplicative step = 1, skipping")
               fails = .true.;  return
            else if (i1 == 0 .or. i2 == 0) then
```

```
              call msg_error ("Scan: " &
                   // "boundary for multiplicative step is zero, skipping")
              fails = .true.;  return
           end if
        end select
     end select
  end function bounds_check_fails
end subroutine cmd_scan_execute
```

Conditionals

Conditionals are implemented as a list that is compiled and evaluated recursively; this allows for a straightforward representation of else if constructs. A cmd_if_t object can hold either an else_if clause which is another object of this type, or an else_body, but not both.

If- or else-bodies are no scoping units, so all data remain global and no copy-in copy-out is needed.

⟨*Commands: types*⟩+≡
```
    type :: cmd_if_t
       private
       type(eval_tree_t) :: if_lexpr
       type(command_list_t), pointer :: if_body => null ()
       type(cmd_if_t), dimension(:), pointer :: elsif_cond => null ()
       type(command_list_t), pointer :: else_body => null ()
    end type cmd_if_t
```

Finalizer.

⟨*Commands: procedures*⟩+≡
```
    recursive subroutine cmd_if_final (cond)
      type(cmd_if_t), intent(inout) :: cond
      integer :: i
      call eval_tree_final (cond%if_lexpr)
      if (associated (cond%if_body)) then
         call command_list_final (cond%if_body)
         deallocate (cond%if_body)
      end if
      if (associated (cond%elsif_cond)) then
         do i = 1, size (cond%elsif_cond)
            call cmd_if_final (cond%elsif_cond(i))
         end do
         deallocate (cond%elsif_cond)
      end if
      if (associated (cond%else_body)) then
         call command_list_final (cond%else_body)
         deallocate (cond%else_body)
      end if
    end subroutine cmd_if_final
```

Output.

⟨*Commands: procedures*⟩+≡
```
    recursive subroutine cmd_if_write (cond, unit, indent)
```

```
      type(cmd_if_t), intent(in) :: cond
      integer, intent(in), optional :: unit, indent
      integer :: u, ind, i
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(A)", advance="no") "if "
      call eval_tree_write (cond%if_lexpr, unit=unit)
      call write_indent (u, indent)
      write (u, "(A)") " then"
      ind = 2;  if (present (indent))  ind = ind + indent
      if (associated (cond%if_body)) then
         call write_indent (u, ind)
         call command_list_write (cond%if_body, unit, ind)
      end if
      if (associated (cond%elsif_cond)) then
         do i = 1, size (cond%elsif_cond)
            if (associated (cond%elsif_cond(i)%if_body)) then
               call write_indent (u, indent)
               write (u, "(A)", advance="no") "elsif "
               call eval_tree_write (cond%elsif_cond(i)%if_lexpr, unit=unit)
               call write_indent (u, indent)
               write (u, "(A)") " then"
               call write_indent (u, ind)
               call command_list_write (cond%elsif_cond(i)%if_body, unit, ind)
            end if
         end do
      end if
      if (associated (cond%else_body)) then
         call write_indent (u, indent)
         write (u, "(A)", advance="no") "else"
         write (u, "(1x,A,1x)") "else"
         call write_indent (u, ind)
         call command_list_write (cond%else_body, unit, ind)
         call write_indent (u, ind)
       else
         write (u, *)
      end if
    end subroutine cmd_if_write
```

Compile the conditional.

⟨*Commands: procedures*⟩+≡
```
    recursive subroutine cmd_if_compile (cond, pn, global)
      type(cmd_if_t), pointer :: cond
      type(parse_node_t), intent(in), target :: pn
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_lexpr, pn_body
      type(parse_node_t), pointer :: pn_elsif_clauses, pn_cmd_elsif
      type(parse_node_t), pointer :: pn_else_clause, pn_cmd_else
      integer :: i, n_elsif
      allocate (cond)
      pn_lexpr => parse_node_get_sub_ptr (pn, 2)
      call eval_tree_init_lexpr (cond%if_lexpr, pn_lexpr, global%var_list)
      pn_body => parse_node_get_next_ptr (pn_lexpr, 2)
      select case (char (parse_node_get_rule_key (pn_body)))
```

```
     case ("command_list")
        allocate (cond%if_body)
        call command_list_compile (cond%if_body, pn_body, global)
        pn_elsif_clauses => parse_node_get_next_ptr (pn_body)
     case default
        pn_elsif_clauses => pn_body
     end select
     select case (char (parse_node_get_rule_key (pn_elsif_clauses)))
     case ("elsif_clauses")
        n_elsif = parse_node_get_n_sub (pn_elsif_clauses)
        allocate (cond%elsif_cond (n_elsif))
        pn_cmd_elsif => parse_node_get_sub_ptr (pn_elsif_clauses)
        do i = 1, n_elsif
           pn_lexpr => parse_node_get_sub_ptr (pn_cmd_elsif, 2)
           call eval_tree_init_lexpr &
                (cond%elsif_cond(i)%if_lexpr, pn_lexpr, global%var_list)
           pn_body => parse_node_get_next_ptr (pn_lexpr, 2)
           if (associated (pn_body)) then
              allocate (cond%elsif_cond(i)%if_body)
              call command_list_compile &
                   (cond%elsif_cond(i)%if_body, pn_body, global)
           end if
           pn_cmd_elsif => parse_node_get_next_ptr (pn_cmd_elsif)
        end do
        pn_else_clause => parse_node_get_next_ptr (pn_elsif_clauses)
     case default
        pn_else_clause => pn_elsif_clauses
     end select
     select case (char (parse_node_get_rule_key (pn_else_clause)))
     case ("else_clause")
        pn_cmd_else => parse_node_get_sub_ptr (pn_else_clause)
        pn_body => parse_node_get_sub_ptr (pn_cmd_else, 2)
        if (associated (pn_body)) then
           allocate (cond%else_body)
           call command_list_compile (cond%else_body, pn_body, global)
        end if
     end select
  end subroutine cmd_if_compile
```

(Recursively) execute the condition. Context remains global in all cases.

⟨*Commands: procedures*⟩+≡
```
  recursive subroutine cmd_if_execute (cond, global)
    type(cmd_if_t), intent(inout), target :: cond
    type(rt_data_t), intent(inout), target :: global
    integer :: i
    call eval_tree_evaluate (cond%if_lexpr)
    if (eval_tree_result_is_known (cond%if_lexpr)) then
       if (eval_tree_get_log (cond%if_lexpr)) then
          if (associated (cond%if_body)) then
             call command_list_execute (cond%if_body, global)
          end if
          return
       end if
    else
```

```
          call error_undecided ()
          return
       end if
     if (associated (cond%elsif_cond)) then
          SCAN_ELSIF: do i = 1, size (cond%elsif_cond)
             call eval_tree_evaluate (cond%elsif_cond(i)%if_lexpr)
             if (eval_tree_result_is_known (cond%elsif_cond(i)%if_lexpr)) then
                if (eval_tree_get_log (cond%elsif_cond(i)%if_lexpr)) then
                   if (associated (cond%elsif_cond(i)%if_body)) then
                      call command_list_execute &
                           (cond%elsif_cond(i)%if_body, global)
                   end if
                   return
                end if
             else
                call error_undecided ()
                return
             end if
          end do SCAN_ELSIF
       end if
     if (associated (cond%else_body)) then
          call command_list_execute (cond%else_body, global)
       end if
   contains
     subroutine error_undecided ()
       call msg_error ("Undefined result of conditional expression: " &
            // "neither branch will be executed")
     end subroutine error_undecided
   end subroutine cmd_if_execute
```

Include another command-list file

The include command allocates a local parse tree. This must not be deleted before the command object itself is deleted, since pointers may point to subobjects of it.

⟨*Commands: types*⟩+≡
```
   type :: cmd_include_t
     private
     type(string_t) :: file
     type(command_list_t), pointer :: command_list => null ()
     type(parse_tree_t) :: parse_tree
   end type cmd_include_t
```

Finalizer: delete the command list.

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_include_final (include)
     type(cmd_include_t), intent(inout) :: include
     call parse_tree_final (include%parse_tree)
     if (associated (include%command_list)) then
        call command_list_final (include%command_list)
        deallocate (include%command_list)
     end if
```

```
      end subroutine cmd_include_final
```

Output: Write file contents as if they were part of the present file (but indented).

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_include_write (include, unit, indent)
      type(cmd_include_t), intent(in) :: include
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(A)")  "! begin include file " &
           // '"' // char (include%file) // '"'
      call command_list_write (include%command_list, unit, indent)
      call write_indent (u, indent)
      write (u, "(A)")  "! end include file " &
           // '"' // char (include%file) // '"'
    end subroutine cmd_include_write
```

Compile file contents: First parse the file, then immediately compile its contents.
Use the global data set.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_include_compile (include, pn, global)
      type(cmd_include_t), pointer :: include
      type(parse_node_t), intent(in), target :: pn
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_arg, pn_file
      type(string_t) :: file
      logical :: exist
      integer :: u
      type(stream_t), target :: stream
      type(lexer_t) :: lexer
      pn_arg => parse_node_get_sub_ptr (pn, 2)
      pn_file => parse_node_get_sub_ptr (pn_arg)
      allocate (include)
      file = parse_node_get_string (pn_file)
      inquire (file=char(file), exist=exist)
      if (exist) then
         include%file = file
      else
         include%file = global%os_data%whizard_cutspath // "/" // file
         inquire (file=char(include%file), exist=exist)
         if (.not. exist) then
            call msg_error ("Include file '" // char (file) // "' not found")
            return
         end if
      end if
      u = free_unit ()
      call lexer_init_cmd_list (lexer, global%lexer)
      call stream_init (stream, char (include%file))
      call lexer_assign_stream (lexer, stream)
      call parse_tree_init (include%parse_tree, syntax_cmd_list, lexer)
      call stream_final (stream)
      call lexer_final (lexer)
```

```
    close (u)
    allocate (include%command_list)
    call command_list_compile &
         (include%command_list, parse_tree_get_root_ptr (include%parse_tree), &
          global)
  end subroutine cmd_include_compile
```

Execute file contents in the global context.

⟨*Commands: procedures*⟩+≡
```
  subroutine cmd_include_execute (include, global)
    type(cmd_include_t), intent(inout), target :: include
    type(rt_data_t), intent(inout), target :: global
    if (associated (include%command_list)) then
       call msg_message &
            ("Including script file '" // char (include%file) // "'")
       call command_list_execute (include%command_list, global)
    end if
  end subroutine cmd_include_execute
```

Quit command execution

The code is the return code of the whole program if it is terminated by this command.

⟨*Commands: types*⟩+≡
```
  type :: cmd_quit_t
     private
     logical :: has_code = .false.
     type(eval_tree_t) :: code_expr
  end type cmd_quit_t
```

Finalizer.

⟨*Commands: procedures*⟩+≡
```
  subroutine cmd_quit_final (quit)
    type(cmd_quit_t), intent(inout) :: quit
    call eval_tree_final (quit%code_expr)
  end subroutine cmd_quit_final
```

Output.

⟨*Commands: procedures*⟩+≡
```
  subroutine cmd_quit_write (quit, unit, indent)
    type(cmd_quit_t), intent(in) :: quit
    integer, intent(in), optional :: unit, indent
    integer :: u
    u = output_unit (unit);  if (u < 0)  return
    call write_indent (u, indent)
    write (u, "(A)")  "quit"
  end subroutine cmd_quit_write
```

Compile: allocate a `quit` object which serves as a placeholder.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_quit_compile (quit, pn, global)
      type(cmd_quit_t), pointer :: quit
      type(parse_node_t), intent(in), target :: pn
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_arg, pn_expr
      allocate (quit)
      pn_arg => parse_node_get_sub_ptr (pn, 2)
      if (associated (pn_arg)) then
         pn_expr => parse_node_get_sub_ptr (pn_arg)
         call eval_tree_init_expr (quit%code_expr, pn_expr, global%var_list)
         quit%has_code = .true.
      end if
    end subroutine cmd_quit_compile
```

Execute: The quit command does not execute anything, it just stops command execution. This is achieved by setting quit flag and quit code in the global variable list. However, the return code, if present, is an expression which has to be evaluated.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_quit_execute (quit, global)
      type(cmd_quit_t), intent(inout), target :: quit
      type(rt_data_t), intent(inout), target :: global
      if (quit%has_code) then
         call eval_tree_evaluate  (quit%code_expr)
         if (eval_tree_result_is_known (quit%code_expr)) then
            global%quit_code = eval_tree_get_int (quit%code_expr)
         else
            call msg_error ("Undefined return code of quit/exit command")
         end if
      end if
      global%quit = .true.
    end subroutine cmd_quit_execute
```

16.1.9 The command list

The command list holds a list of commands and relevant global data.

⟨*Commands: public*⟩+≡
```
    public :: command_list_t
```

⟨*Commands: types*⟩+≡
```
    type :: command_list_t
      private
      type(command_t), pointer :: first => null ()
      type(command_t), pointer :: last => null ()
    end type command_list_t
```

Append a new command.

⟨*Commands: procedures*⟩+≡
```
    subroutine command_list_append (cmd_list, command)
```

```
      type(command_list_t), intent(inout) :: cmd_list
      type(command_t), intent(in), target :: command
      if (associated (cmd_list%last)) then
         cmd_list%last%next => command
      else
         cmd_list%first => command
      end if
      cmd_list%last => command
   end subroutine command_list_append
```

Finalize.

⟨*Commands: public*⟩+≡
```
   public :: command_list_final
```

⟨*Commands: procedures*⟩+≡
```
   recursive subroutine command_list_final (cmd_list)
      type(command_list_t), intent(inout) :: cmd_list
      type(command_t), pointer :: command
      do while (associated (cmd_list%first))
         command => cmd_list%first
         cmd_list%first => cmd_list%first%next
         call command_final (command)
         deallocate (command)
      end do
      cmd_list%last => null ()
   end subroutine command_list_final
```

Output.

⟨*Commands: public*⟩+≡
```
   public :: command_list_write
```

⟨*Commands: procedures*⟩+≡
```
   recursive subroutine command_list_write (cmd_list, unit, indent)
      type(command_list_t), intent(in) :: cmd_list
      integer, intent(in), optional :: unit, indent
      type(command_t), pointer :: command
      command => cmd_list%first
      do while (associated (command))
         if (present (indent)) then
            call command_write (command, unit, indent + 1)
         else
            call command_write (command, unit, 1)
         end if
         command => command%next
      end do
   end subroutine command_list_write
```

16.1.10   Compiling the parse tree

Transform a parse tree into a command list. Initialization is assumed to be done.

⟨*Commands: public*⟩+≡
```
   public :: command_list_compile
```

⟨*Commands: procedures*⟩+≡
```
    recursive subroutine command_list_compile (cmd_list, pn, global)
      type(command_list_t), intent(inout), target :: cmd_list
      type(parse_node_t), intent(in), target :: pn
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_cmd
      type(command_t), pointer :: command
      integer :: i
      pn_cmd => parse_node_get_sub_ptr (pn)
      do i = 1, parse_node_get_n_sub (pn)
         call command_compile (command, pn_cmd, global)
         call command_list_append (cmd_list, command)
         pn_cmd => parse_node_get_next_ptr (pn_cmd)
      end do
    end subroutine command_list_compile
```

16.1.11   Executing the command list

⟨*Commands: public*⟩+≡
```
    public :: command_list_execute
```

⟨*Commands: procedures*⟩+≡
```
    recursive subroutine command_list_execute (cmd_list, global, print)
      type(command_list_t), intent(in) :: cmd_list
      type(rt_data_t), intent(inout), target :: global
      logical, intent(in), optional :: print
      type(command_t), pointer :: command
      command => cmd_list%first
      COMMAND_COND: do while (associated (command))
         call command_execute (command, global, print)
         if (global%quit)  exit COMMAND_COND
         command => command%next
      end do COMMAND_COND
    end subroutine command_list_execute
```

16.1.12   Command list syntax

⟨*Commands: public*⟩+≡
```
    public :: syntax_cmd_list
```

⟨*Commands: variables*⟩≡
```
    type(syntax_t), target, save :: syntax_cmd_list
```

⟨*Commands: public*⟩+≡
```
    public :: syntax_cmd_list_init
```

⟨*Commands: procedures*⟩+≡
```
    subroutine syntax_cmd_list_init ()
      type(ifile_t) :: ifile
      call define_cmd_list_syntax (ifile)
      call syntax_init (syntax_cmd_list, ifile)
      call ifile_final (ifile)
```

```
          end subroutine syntax_cmd_list_init


⟨Commands: public⟩+≡
      public :: syntax_cmd_list_final

⟨Commands: procedures⟩+≡
      subroutine syntax_cmd_list_final ()
        call syntax_final (syntax_cmd_list)
      end subroutine syntax_cmd_list_final


⟨Commands: public⟩+≡
      public :: syntax_cmd_list_write

⟨Commands: procedures⟩+≡
      subroutine syntax_cmd_list_write (unit)
        integer, intent(in), optional :: unit
        call syntax_write (syntax_cmd_list, unit)
      end subroutine syntax_cmd_list_write


⟨Commands: procedures⟩+≡
      subroutine define_cmd_list_syntax (ifile)
        type(ifile_t), intent(inout) :: ifile
        call ifile_append (ifile, "SEQ command_list = command*")
        call ifile_append (ifile, "ALT command = " &
             // "cmd_model | cmd_library | cmd_iterations | cmd_sample_format | " &
             // "cmd_seed | " &
             // "cmd_var | cmd_slha | " &
             // "cmd_print | cmd_printf | cmd_show | cmd_echo | " &
             // "cmd_expect | " &
             // "cmd_cuts | cmd_weight | cmd_scale | " &
             // "cmd_beams | cmd_integrate | " &
             // "cmd_observable | cmd_histogram | cmd_plot | cmd_clear | " &
             // "cmd_record | " &
             // "cmd_analysis | cmd_write_analysis | " &
             // "cmd_unstable | cmd_stable | cmd_simulate | " &
             // "cmd_process | cmd_compile | cmd_load | cmd_exec | " &
             // "cmd_scan | cmd_if | cmd_include | cmd_quit")
        call ifile_append (ifile, "GRO options = '{' local_command_list '}'")
        call ifile_append (ifile, "SEQ local_command_list = local_command*")
        call ifile_append (ifile, "ALT local_command = " &
             // "cmd_model | cmd_library | cmd_iterations | cmd_sample_format | " &
             // "cmd_seed | " &
             // "cmd_var | cmd_slha | " &
             // "cmd_print | cmd_printf | cmd_show | cmd_echo | " &
             // "cmd_expect | " &
             // "cmd_cuts | cmd_weight | cmd_scale | " &
             // "cmd_beams | " &
             // "cmd_observable | cmd_histogram | cmd_plot | cmd_clear | " &
             // "cmd_record | " &
             // "cmd_analysis | cmd_write_analysis")
        call ifile_append (ifile, "SEQ cmd_model = model '=' model_name")
        call ifile_append (ifile, "KEY model")
        call ifile_append (ifile, "ALT model_name = model_id | string_literal")
        call ifile_append (ifile, "IDE model_id")
```

```
call ifile_append (ifile, "SEQ cmd_library = library '=' lib_name")
call ifile_append (ifile, "KEY library")
call ifile_append (ifile, "ALT lib_name = lib_id | string_literal")
call ifile_append (ifile, "IDE lib_id")
call ifile_append (ifile, "ALT cmd_var = " &
     // "cmd_log_decl | cmd_log | " &
     // "cmd_int | cmd_real | cmd_complex | cmd_num | " &
     // "cmd_string_decl | cmd_string | cmd_alias")
call ifile_append (ifile, "SEQ cmd_log_decl = logical cmd_log")
call ifile_append (ifile, "SEQ cmd_log = '?' var_name '=' lexpr")
call ifile_append (ifile, "SEQ cmd_int = int var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_real = real var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_complex = complex var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_num = var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_string_decl = string cmd_string")
call ifile_append (ifile, "SEQ cmd_string = " &
     // "'$' var_name '=' sexpr") ! $
call ifile_append (ifile, "SEQ cmd_alias = alias var_name '=' cexpr")
call ifile_append (ifile, "SEQ cmd_slha = slha_action slha_arg options?")
call ifile_append (ifile, "ALT slha_action = " &
     // "read_slha | write_slha")
call ifile_append (ifile, "KEY read_slha")
call ifile_append (ifile, "KEY write_slha")
call ifile_append (ifile, "ARG slha_arg = ( string_literal )")
call ifile_append (ifile, "SEQ cmd_print = print_cmd options?")
call ifile_append (ifile, "SEQ print_cmd = print sprintf_args?")
call ifile_append (ifile, "KEY print")
call ifile_append (ifile, "SEQ cmd_printf = printf_cmd options?")
call ifile_append (ifile, "SEQ printf_cmd = printf_clause sprintf_args?")
call ifile_append (ifile, "SEQ printf_clause = printf sexpr")
call ifile_append (ifile, "KEY printf")
call ifile_append (ifile, "SEQ cmd_show = show show_arg?")
call ifile_append (ifile, "KEY show")
call ifile_append (ifile, "ARG show_arg = ( var_generic* )")
call ifile_append (ifile, "ALT var_generic = " &
     // "model | beams | results | unstable | real | int | " &
     // "cuts | weight | scale | analysis | expect | " &
     // "library_spec | " &
     // "intrinsic | result_var | " &
     // "log_var | alias_var | string_var | num_var")
call ifile_append (ifile, "KEY results")
call ifile_append (ifile, "KEY intrinsic")
call ifile_append (ifile, "SEQ library_spec = library lib_name?")
call ifile_append (ifile, "SEQ result_var = result_key result_arg?")
call ifile_append (ifile, "SEQ log_var = '?' log_var_spec?")
call ifile_append (ifile, "ALT log_var_spec = log_val | variable")
call ifile_append (ifile, "GRO log_val  = ( lexpr )")
call ifile_append (ifile, "SEQ alias_var = alias alias_var_spec?")
call ifile_append (ifile, "ALT alias_var_spec = alias_val | variable")
call ifile_append (ifile, "GRO alias_val  = ( cexpr )")
call ifile_append (ifile, "SEQ string_var = '$' string_var_spec?")  ! $
call ifile_append (ifile, "ALT string_var_spec = string_val | variable")
call ifile_append (ifile, "GRO string_val  = ( sexpr )")
call ifile_append (ifile, "SEQ num_var = num_var_spec")
```

1288

```
call ifile_append (ifile, "ALT num_var_spec = num_val | variable")
call ifile_append (ifile, "GRO num_val  = ( expr )")
call ifile_append (ifile, "SEQ cmd_expect = expect expect_arg options?")
call ifile_append (ifile, "KEY expect")
call ifile_append (ifile, "ARG expect_arg = ( lexpr )")
call ifile_append (ifile, "SEQ cmd_cuts = cuts '=' lexpr")
call ifile_append (ifile, "SEQ cmd_echo = echo echo_arg?")
call ifile_append (ifile, "KEY echo")
call ifile_append (ifile, "ARG echo_arg = ( sexpr* )")
call ifile_append (ifile, "SEQ cmd_weight = weight '=' expr")
call ifile_append (ifile, "SEQ cmd_scale = scale '=' expr")
call ifile_append (ifile, "KEY cuts")
call ifile_append (ifile, "KEY weight")
call ifile_append (ifile, "KEY scale")
call ifile_append (ifile, "SEQ cmd_process = process process_id '=' " &
     // "prt_list '=>' prt_list options?")
call ifile_append (ifile, "KEY process")
call ifile_append (ifile, "KEY '=>'")
call ifile_append (ifile, "LIS prt_list = cexpr+")
call ifile_append (ifile, "SEQ cmd_compile = compile_cmd options?")
call ifile_append (ifile, "SEQ compile_cmd = compile_clause compile_arg?")
call ifile_append (ifile, "SEQ compile_clause = compile exec_name_spec?")
call ifile_append (ifile, "KEY compile")
call ifile_append (ifile, "SEQ exec_name_spec = as exec_name")
call ifile_append (ifile, "KEY as")
call ifile_append (ifile, "ALT exec_name = exec_id | string_literal")
call ifile_append (ifile, "IDE exec_id")
call ifile_append (ifile, "ARG compile_arg = ( lib_name* )")
call ifile_append (ifile, "SEQ cmd_load = load load_arg?")
call ifile_append (ifile, "KEY load")
call ifile_append (ifile, "ARG load_arg = ( lib_name* )")
call ifile_append (ifile, "SEQ cmd_exec = exec exec_arg")
call ifile_append (ifile, "KEY exec")
call ifile_append (ifile, "ARG exec_arg = ( sexpr )")
call ifile_append (ifile, "SEQ cmd_beams = beams '=' beam_def")
call ifile_append (ifile, "KEY beams")
call ifile_append (ifile, "SEQ beam_def = beam_spec strfun_seq*")
call ifile_append (ifile, "SEQ beam_spec = beam_list options?")
call ifile_append (ifile, "LIS beam_list = cexpr, cexpr?")
call ifile_append (ifile, "SEQ strfun_seq = '=>' strfun_pair")
call ifile_append (ifile, "LIS strfun_pair = strfun_def, strfun_def?")
call ifile_append (ifile, "SEQ strfun_def = strfun_id options?")
call ifile_append (ifile, "ALT strfun_id = " &
     // "none | lhapdf | isr | epa | ewa")
call ifile_append (ifile, "KEY none")
call ifile_append (ifile, "KEY lhapdf")
call ifile_append (ifile, "KEY isr")
call ifile_append (ifile, "KEY epa")
call ifile_append (ifile, "KEY ewa")
call ifile_append (ifile, "SEQ cmd_integrate = " &
     // "integrate proc_arg options?")
call ifile_append (ifile, "KEY integrate")
call ifile_append (ifile, "ARG proc_arg = ( proc_id* )")
call ifile_append (ifile, "IDE proc_id")
```

```
call ifile_append (ifile, "SEQ cmd_seed = seed '=' expr")
call ifile_append (ifile, "KEY seed")
call ifile_append (ifile, "SEQ cmd_iterations = " &
     // "iterations '=' iterations_list")
call ifile_append (ifile, "KEY iterations")
call ifile_append (ifile, "LIS iterations_list = iterations_spec+")
call ifile_append (ifile, "ALT iterations_spec = it_spec | calls_spec")
call ifile_append (ifile, "SEQ it_spec = expr calls_spec?")
call ifile_append (ifile, "SEQ calls_spec = ':' expr")
call ifile_append (ifile, "SEQ cmd_sample_format = " &
     // "sample_format '=' event_format_list")
call ifile_append (ifile, "KEY sample_format")
call ifile_append (ifile, "LIS event_format_list = event_format+")
call ifile_append (ifile, "IDE event_format")
call ifile_append (ifile, "SEQ cmd_observable = " &
     // "observable analysis_tag options?")
call ifile_append (ifile, "KEY observable")
call ifile_append (ifile, "SEQ cmd_histogram = " &
     // "histogram analysis_tag histogram_arg options?")
call ifile_append (ifile, "KEY histogram")
call ifile_append (ifile, "ARG histogram_arg = (expr, expr, expr)")
call ifile_append (ifile, "SEQ cmd_plot = " &
     // "plot analysis_tag plot_arg options?")
call ifile_append (ifile, "KEY plot")
call ifile_append (ifile, "ARG plot_arg = (expr, expr)")
call ifile_append (ifile, "SEQ cmd_analysis = analysis '=' lexpr")
call ifile_append (ifile, "KEY analysis")
call ifile_append (ifile, "SEQ cmd_write_analysis = " &
     // "write_analysis write_analysis_arg?")
call ifile_append (ifile, "KEY write_analysis")
call ifile_append (ifile, "ARG write_analysis_arg = ( analysis_tag* )")
call ifile_append (ifile, "SEQ cmd_clear = clear clear_arg?")
call ifile_append (ifile, "KEY clear")
call ifile_append (ifile, "ARG clear_arg = ( clear_obj* )")
call ifile_append (ifile, "ALT clear_obj = " &
     // "iterations | cuts | weight | scale | analysis | expect | " &
     // "analysis_tag")
call ifile_append (ifile, "SEQ cmd_record = record_cmd")
call ifile_append (ifile, "SEQ cmd_unstable = " &
     // "unstable unstable_list options?")
call ifile_append (ifile, "KEY unstable")
call ifile_append (ifile, "LIS unstable_list = unstable_decl+")
call ifile_append (ifile, "SEQ unstable_decl = cexpr unstable_arg")
call ifile_append (ifile, "ARG unstable_arg = ( proc_id+ )")
call ifile_append (ifile, "SEQ cmd_stable = stable stable_list options?")
call ifile_append (ifile, "KEY stable")
call ifile_append (ifile, "LIS stable_list = cexpr+")
call ifile_append (ifile, "SEQ cmd_simulate = " &
     // "simulate proc_arg options?")
call ifile_append (ifile, "KEY simulate")
call ifile_append (ifile, "SEQ cmd_scan = scan_spec scan_body?")
call ifile_append (ifile, "SEQ scan_spec = scan scan_command?")
call ifile_append (ifile, "KEY scan")
call ifile_append (ifile, "ALT scan_command = " &
```

```
            // "cmd_model_list | cmd_library_list | " &
            // "cmd_seed_list | " &
            // "cmd_cuts_list | cmd_weight_list | cmd_scale_list | " &
            // "cmd_analysis_list | " &
            // "cmd_var_list")
      call ifile_append (ifile, "SEQ cmd_model_list = model = model_list_arg")
      call ifile_append (ifile, "ARG model_list_arg = ( model_name* )")
      call ifile_append (ifile, "SEQ cmd_library_list = " &
          // "library = library_list_arg")
      call ifile_append (ifile, "ARG library_list_arg = ( lib_name* )")
      call ifile_append (ifile, "SEQ cmd_seed_list = seed = num_step_list_arg")
      call ifile_append (ifile, "ARG num_step_list_arg = ( num_steps* )")
      call ifile_append (ifile, "SEQ num_steps = expr range_spec?")
      call ifile_append (ifile, "SEQ range_spec = '=>' expr step_spec?")
      call ifile_append (ifile, "SEQ step_spec = step_op expr")
      call ifile_append (ifile, "ALT step_op = '/+' | '/-' | '/*' | '//'")
      call ifile_append (ifile, "KEY '/+'")
      call ifile_append (ifile, "KEY '/-'")
      call ifile_append (ifile, "KEY '/*'")
!     call ifile_append (ifile, "KEY '//'")
      call ifile_append (ifile, "ALT cmd_var_list = " &
          // "log_decl_list | log_list | " &
          // "int_list | real_list | complex_list | num_list | " &
          // "string_decl_list | string_list | alias_list")
      call ifile_append (ifile, "SEQ log_decl_list = logical log_list")
      call ifile_append (ifile, "SEQ log_list = '?' var_name '=' log_list_arg")
      call ifile_append (ifile, "ARG log_list_arg = ( lexpr* )")
      call ifile_append (ifile, "SEQ int_list = " &
          // "int var_name '=' num_step_list_arg")
      call ifile_append (ifile, "SEQ real_list = " &
          // "real var_name '=' num_step_list_arg")
      call ifile_append (ifile, "SEQ complex_list = " &
          // "complex var_name '=' num_step_list_arg")
      call ifile_append (ifile, "SEQ num_list = var_name '=' num_step_list_arg")
      call ifile_append (ifile, "SEQ string_decl_list = string string_list")
      call ifile_append (ifile, "SEQ string_list = " &
          // "'$' var_name '=' string_list_arg") !$
      call ifile_append (ifile, "ARG string_list_arg = ( sexpr* )")
      call ifile_append (ifile, "SEQ alias_list = " &
          // "alias var_name alias_list_arg")
      call ifile_append (ifile, "ARG alias_list_arg = ( cexpr* )")
      call ifile_append (ifile, "SEQ cmd_cuts_list = cuts = log_list_arg")
      call ifile_append (ifile, "SEQ cmd_weight_list = weight = num_list_arg")
      call ifile_append (ifile, "SEQ cmd_scale_list = scale = num_list_arg")
      call ifile_append (ifile, "ARG num_list_arg = ( expr* )")
      call ifile_append (ifile, "SEQ cmd_analysis_list = analysis = log_list_arg")
      call ifile_append (ifile, "GRO scan_body = '{' command_list '}'")
      call ifile_append (ifile, "SEQ cmd_if = " &
          // "if lexpr then command_list elsif_clauses else_clause endif")
      call ifile_append (ifile, "SEQ elsif_clauses = cmd_elsif*")
      call ifile_append (ifile, "SEQ cmd_elsif = elsif lexpr then command_list")
      call ifile_append (ifile, "KEY elsif")
      call ifile_append (ifile, "SEQ else_clause = cmd_else?")
      call ifile_append (ifile, "SEQ cmd_else = else command_list")
```

```
    call ifile_append (ifile, "SEQ cmd_include = include include_arg")
    call ifile_append (ifile, "KEY include")
    call ifile_append (ifile, "ARG include_arg = ( string_literal )")
    call ifile_append (ifile, "SEQ cmd_quit = quit_cmd quit_arg?")
    call ifile_append (ifile, "ALT quit_cmd = quit | exit")
    call ifile_append (ifile, "KEY quit")
    call ifile_append (ifile, "KEY exit")
    call ifile_append (ifile, "ARG quit_arg = ( expr )")
    call define_expr_syntax (ifile, particles=.true., analysis=.true.)
  end subroutine define_cmd_list_syntax
```

⟨*Commands: public*⟩+≡
```
    public :: lexer_init_cmd_list
```

⟨*Commands: procedures*⟩+≡
```
    subroutine lexer_init_cmd_list (lexer, parent_lexer)
      type(lexer_t), intent(out) :: lexer
      type(lexer_t), intent(in), optional, target :: parent_lexer
      call lexer_init (lexer, &
           comment_chars = "#!", &
           quote_chars = '"', &
           quote_match = '"', &
           single_chars = "()[]{},:&%?$@", &
           special_class = (/ "+-*/^", "<>=~ " /) , &
           keyword_list = syntax_get_keyword_list_ptr (syntax_cmd_list), &
           parent = parent_lexer)
    end subroutine lexer_init_cmd_list
```

16.1.13 Test

⟨*Commands: public*⟩+≡
```
    public :: command_test
```

⟨*Commands: procedures*⟩+≡
```
    subroutine command_test ()
      integer :: u
      type(stream_t), target :: stream
      type(lexer_t) :: lexer
      type(parse_tree_t) :: parse_tree
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      print *, "* Initialization"
      call os_data_init (global%os_data)
      global%os_data%fcflags = "-gline -C=all"
      allocate (global%rng)
      call tao_random_create (global%rng, 0)
      call syntax_model_file_init ()
      call syntax_phs_forest_init ()
      call syntax_pexpr_init ()
      call syntax_cmd_list_init ()
      call lexer_init_cmd_list (lexer)
      print *, "* Open 'whizard.sin'"
      u = free_unit ()
```

```
      open (unit=u, file="whizard.sin")
      call stream_init (stream, u)
      print *, "* Parse"
      call lexer_assign_stream (lexer, stream)
      call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
      call stream_final (stream)
      close (u)
      call parse_tree_write (parse_tree)
      print *
      print *, "* Compile command list"
      if (associated (parse_tree_get_root_ptr (parse_tree))) then
         call command_list_compile &
              (command_list, parse_tree_get_root_ptr (parse_tree), global)
      end if
      print *, "* Command list:"
      call command_list_write (command_list)
      print *
      print *, "* Execute command list"
      call command_list_execute (command_list, global, print=.true.)
      print *
      print *, "* Cleanup"
      call command_list_final (command_list)
      call process_store_final ()
      call model_list_final ()
      call syntax_cmd_list_final ()
      call syntax_pexpr_final ()
      call syntax_phs_forest_final ()
      call syntax_model_file_final ()
    end subroutine command_test
```

16.2 Toplevel module WHIZARD

⟨whizard.f90⟩≡
  ⟨*File header*⟩

```
  module whizard
```

  ⟨*Use file utils*⟩
  ⟨*Use strings*⟩
```
    use limits, only: VERSION_STRING !NODEP!
    use limits, only: EOF, BACKSLASH !NODEP!
    use diagnostics !NODEP!
    use formats
    use md5
    use os_interface
    use lexers
    use parser
    use colors
    use state_matrices
    use analysis
    use variables
    use expressions
```

```
     use models
     use evaluators
     use phs_forests
     use hard_interactions
     use processes
     use decays
     use process_libraries
     use slha_interface
     use commands
     use vamp !NODEP!
```

⟨*Standard module head*⟩

⟨*WHIZARD: public*⟩

⟨*WHIZARD: variables*⟩

```
   save

contains
```

⟨*WHIZARD: procedures*⟩

```
end module whizard
```

16.2.1 Initialization and finalization

These procedures initialize and finalize global variables. Most of them are collected in the `global` data record located here, the others are syntax tables located in various modules, which do not change during program execution. Furthermore, there is a global model list and a global process store, which get filled during program execution but are finalized here.

During initialization, we can preload a default model and initialize a default library for setting up processes. The default library is loaded if requested by the setup. Further libraries can be loaded as specified by command-line flags.

⟨*WHIZARD: variables*⟩≡
```
    type(rt_data_t), target :: global
```

⟨*WHIZARD: public*⟩≡
```
    public :: whizard_init
```

⟨*WHIZARD: procedures*⟩≡
```
    subroutine whizard_init &
        (preload_model, preload_libs, default_lib, &
         rebuild_library, rebuild_phs, rebuild_grids, rebuild_events, &
         recompile_library, &
         time_estimate, &
         paths)
      type(string_t), intent(in) :: preload_model, preload_libs
      type(string_t), intent(in) :: default_lib
      logical, intent(in) :: rebuild_library, rebuild_phs, rebuild_grids
      logical, intent(in) :: rebuild_events
      logical, intent(in) :: recompile_library
      logical, intent(in) :: time_estimate
```

```
type(paths_t), intent(in), optional :: paths
type(string_t) :: filename, libname, libs
type(var_list_t), pointer :: model_vars
call rt_data_global_init (global, paths)
call var_list_append_log &
     (global%var_list, var_str ("?rebuild_library"), rebuild_library, &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?rebuild_phase_space"), rebuild_phs, &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?rebuild_grids"), rebuild_grids, &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?rebuild_events"), rebuild_events, &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?recompile_library"), recompile_library, &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?time_estimate"), time_estimate, &
      intrinsic=.true.)
call init_syntax_tables ()
call process_library_store_load_static &
     (global%os_data, global%prc_lib, global%model, global%var_list)
libs = adjustl (preload_libs)
SCAN_LIBS: do while (libs /= "")
   call split (libs, libname, " ")
   call process_library_store_append &
        (libname, global%os_data, global%prc_lib)
   call process_library_load &
        (global%prc_lib, global%os_data, global%model, global%var_list, &
         ignore=.true.)
end do SCAN_LIBS
if (.not. associated (global%prc_lib)) then
   call process_library_store_append &
        (default_lib, global%os_data, global%prc_lib)
   if (.not. (rebuild_library .or. recompile_library)) then
      call process_library_load (global%prc_lib, &
           global%os_data, global%model, global%var_list, ignore=.true.)
   else
      call var_list_set_string (global%var_list, &
           var_str ("$library_name"), &
           process_library_get_name (global%prc_lib), is_known=.true.)
   end if
end if
filename = preload_model // ".mdl"
call model_list_read_model &
     (preload_model, filename, global%os_data, global%model)
if (associated (global%model)) then
   model_vars => model_get_var_list_ptr (global%model)
   call var_list_init_copies (global%var_list, model_vars)
   call var_list_synchronize &
        (global%var_list, model_vars, reset_pointers = .true.)
```

```
        call msg_message ("Using model: " &
            // char (model_get_name (global%model)))
        call var_list_set_string (global%var_list, var_str ("$model_name"), &
            model_get_name (global%model), is_known=.true.)
    end if
  end subroutine whizard_init
```

Initialize/finalize the syntax tables used by WHIZARD:

⟨*WHIZARD: public*⟩+≡
```
    public :: init_syntax_tables
    public :: final_syntax_tables
```

⟨*WHIZARD: procedures*⟩+≡
```
  subroutine init_syntax_tables ()
    call syntax_model_file_init ()
    call syntax_phs_forest_init ()
    call syntax_pexpr_init ()
    call syntax_slha_init ()
    call syntax_cmd_list_init ()
  end subroutine init_syntax_tables

  subroutine final_syntax_tables ()
    call syntax_model_file_final ()
    call syntax_phs_forest_final ()
    call syntax_pexpr_final ()
    call syntax_slha_final ()
    call syntax_cmd_list_final ()
  end subroutine final_syntax_tables
```

Write the syntax tables to external files.

⟨*WHIZARD: public*⟩+≡
```
    public :: write_syntax_tables
```

⟨*WHIZARD: procedures*⟩+≡
```
  subroutine write_syntax_tables ()
    integer :: unit
    character(*), parameter :: file_model = "whizard.model_file.syntax"
    character(*), parameter :: file_phs = "whizard.phase_space_file.syntax"
    character(*), parameter :: file_pexpr = "whizard.prt_expressions.syntax"
    character(*), parameter :: file_slha = "whizard.slha.syntax"
    character(*), parameter :: file_sindarin = "whizard.sindarin.syntax"
    unit = free_unit ()
    print *, "Writing file '" // file_model // "'"
    open (unit=unit, file=file_model, status="replace", action="write")
    write (unit, "(A)")  VERSION_STRING
    write (unit, "(A)")  "Syntax definition file: " // file_model
    call syntax_model_file_write (unit)
    close (unit)
    print *, "Writing file '" // file_phs // "'"
    open (unit=unit, file=file_phs, status="replace", action="write")
    write (unit, "(A)")  VERSION_STRING
    write (unit, "(A)")  "Syntax definition file: " // file_phs
    call syntax_phs_forest_write (unit)
    close (unit)
```

```
      print *, "Writing file '" // file_pexpr // "'"
      open (unit=unit, file=file_pexpr, status="replace", action="write")
      write (unit, "(A)")  VERSION_STRING
      write (unit, "(A)")  "Syntax definition file: " // file_pexpr
      call syntax_pexpr_write (unit)
      close (unit)
      print *, "Writing file '" // file_slha // "'"
      open (unit=unit, file=file_slha, status="replace", action="write")
      write (unit, "(A)")  VERSION_STRING
      write (unit, "(A)")  "Syntax definition file: " // file_slha
      call syntax_slha_write (unit)
      close (unit)
      print *, "Writing file '" // file_sindarin // "'"
      open (unit=unit, file=file_sindarin, status="replace", action="write")
      write (unit, "(A)")  VERSION_STRING
      write (unit, "(A)")  "Syntax definition file: " // file_sindarin
      call syntax_cmd_list_write (unit)
      close (unit)
    end subroutine write_syntax_tables
```

Apart from the global data which have been initialized above, the process and model lists need to be finalized.

⟨*WHIZARD: public*⟩+≡
```
    public :: whizard_final
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_final ()
      call rt_data_global_final (global)
      call decay_store_final ()
      call process_store_final ()
      call model_list_final ()
      call final_syntax_tables ()
    end subroutine whizard_final
```

16.2.2   Execute command lists

Process standard input as a command list. The whole input is read, compiled and executed as a whole.

⟨*WHIZARD: public*⟩+≡
```
    public :: whizard_process_stdin
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_process_stdin (quit, quit_code)
      logical, intent(out) :: quit
      integer, intent(out) :: quit_code
      type(lexer_t), target :: lexer
      type(stream_t), target :: stream
      call msg_message ("Reading commands from standard input")
      call lexer_init_cmd_list (lexer)
      call stream_init (stream, 5)
      call whizard_process_stream (stream, lexer, quit, quit_code)
      call stream_final (stream)
```

```
      call lexer_final (lexer)
    end subroutine whizard_process_stdin
```

Process a file as a command list.

⟨*WHIZARD: public*⟩+≡
```
    public :: whizard_process_file
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_process_file (file, quit, quit_code)
      type(string_t), intent(in) :: file
      logical, intent(out) :: quit
      integer, intent(out) :: quit_code
      integer :: u
      type(lexer_t), target :: lexer
      type(stream_t), target :: stream
      logical :: exist
      call msg_message ("Reading commands from file '" // char (file) // "'")
      inquire (file=char(file), exist=exist)
      if (exist) then
         u = free_unit ()
         call lexer_init_cmd_list (lexer)
         call stream_init (stream, char (file))
         call whizard_process_stream (stream, lexer, quit, quit_code)
         call stream_final (stream)
         call lexer_final (lexer)
      else
         call msg_error ("File '" // char (file) // "' not found")
      end if
    end subroutine whizard_process_file
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_process_stream (stream, lexer, quit, quit_code)
      type(stream_t), intent(inout), target :: stream
      type(lexer_t), intent(inout), target :: lexer
      logical, intent(out) :: quit
      integer, intent(out) :: quit_code
      type(parse_tree_t) :: parse_tree
      type(command_list_t), target :: command_list
      global%lexer => lexer
      call lexer_assign_stream (lexer, stream)
      call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
      ! call parse_tree_write (parse_tree)
      if (associated (parse_tree_get_root_ptr (parse_tree))) then
         call command_list_compile &
              (command_list, parse_tree_get_root_ptr (parse_tree), global)
      end if
      ! call command_list_write (command_list)
      ! call command_list_execute (command_list, global, print=.true.)
      call command_list_execute (command_list, global)
      call command_list_final (command_list)
      quit = global%quit
      quit_code = global%quit_code
    end subroutine whizard_process_stream
```

1298

16.2.3 The WHIZARD shell

This procedure implements interactive mode. One line is processed at a time.

⟨*WHIZARD: public*⟩+≡
```
    public :: whizard_shell
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_shell (quit_code)
      integer, intent(out) :: quit_code
      type(lexer_t), target :: lexer
      type(stream_t), target :: stream
      type(string_t) :: prompt1
      type(string_t) :: prompt2
      type(string_t) :: input
      type(string_t) :: extra
      integer :: last
      integer :: iostat
      logical :: mask_tmp
      logical :: quit
      call msg_message ("Launching interactive shell")
      call lexer_init_cmd_list (lexer)
      prompt1 = "whish? "
      prompt2 = "      > "
      COMMAND_LOOP: do
         call put (6, prompt1)
         call get (5, input, iostat=iostat)
         if (iostat > 0 .or. iostat == EOF) exit COMMAND_LOOP
         CONTINUE_INPUT: do
            last = len_trim (input)
            if (extract (input, last, last) /= BACKSLASH)  exit CONTINUE_INPUT
            call put (6, prompt2)
            call get (5, extra, iostat=iostat)
            if (iostat > 0) exit COMMAND_LOOP
            input = replace (input, last, extra)
         end do CONTINUE_INPUT
         call stream_init (stream, input)
         mask_tmp = mask_fatal_errors
         mask_fatal_errors = .true.
         call whizard_process_stream (stream, lexer, quit, quit_code)
         msg_count = 0
         mask_fatal_errors = mask_tmp
         call stream_final (stream)
         if (quit)  exit COMMAND_LOOP
      end do COMMAND_LOOP
      print *
      call lexer_final (lexer)
    end subroutine whizard_shell
```

16.2.4 Self-tests

This is for developers only, but needs a well-defined interface.

⟨*WHIZARD: public*⟩+≡
```
    public :: whizard_check
```

⟨*WHIZARD: procedures*⟩+≡
```
subroutine whizard_check (check)
  type(string_t), intent(in) :: check
  call msg_message (repeat ('=', 76), 0)
  call msg_message ("Running self-test: " // char (check), 0)
  call msg_message (repeat ('-', 76), 0)
  select case (char (check))
  case ("formats");  call format_test ()
  case ("md5");  call md5_test ()
  case ("colors");  call color_test ()
  case ("state_matrices");  call state_matrix_test ()
  case ("analysis");  call analysis_test ()
  case ("expressions");  call expressions_test ()
  case ("hard_interactions");  call hard_interaction_test (global%model)
  case ("evaluators");  call evaluator_test (global%model)
  case ("slha_interface");  call slha_test ()
  case default
     call msg_error ("Self-test '" // char (check) // "' not implemented.")
  end select
end subroutine whizard_check
```

16.3   Driver program

The main program handles command options, initializes the environment, and
runs WHIZARD in a particular mode (interactive, file, standard input).

⟨*Limits: public parameters*⟩+≡
```
integer, parameter, public :: CMDLINE_ARG_LEN = 1000
```

⟨`main.f90`⟩≡
  ⟨*File header*⟩
```

  program main
```
  ⟨*Use strings*⟩
```
  use system_dependencies !NODEP!
  use limits, only: CMDLINE_ARG_LEN !NODEP!
  use diagnostics !NODEP!
  use os_interface
  use whizard

  implicit none

  ! Main program variable declarations
  character(CMDLINE_ARG_LEN) :: arg
  character(2) :: option
  type(string_t) :: long_option, value
  integer :: i, j, arg_len, arg_status
  logical :: look_for_options
  logical :: interactive
  type(string_t) :: files, this, model, libname, library, libraries, logfile
  type(string_t) :: check, checks
  type(paths_t) :: paths
```

```
logical :: rebuild_library, rebuild_phs, rebuild_grids, rebuild_events
logical :: recompile_library
logical :: time_estimate

! Exit status
logical :: quit = .false.
integer :: quit_code = 0

! Initial values
look_for_options = .true.
interactive = .false.
files = ""
model = "SM"
libname = "processes"
library = ""
logfile = "whizard.log"
libraries = ""
check = ""
checks = ""
rebuild_library = .false.
rebuild_phs = .false.
rebuild_grids = .false.
rebuild_events = .false.
recompile_library = .false.
time_estimate = .true.
call paths_init (paths)

! Read and process options
i = 0
SCAN_CMDLINE: do
   i = i + 1
   call get_command_argument (i, arg, arg_len, arg_status)
   select case (arg_status)
   case (0)
   case (-1)
      call msg_error (" Command argument truncated: '" // arg // "'")
   case default
      exit SCAN_CMDLINE
   end select
   if (look_for_options) then
      select case (arg(1:2))
      case ("--")
         value = trim (arg)
         call split (value, long_option, "=")
         select case (char (long_option))
         case ("--version")
            call no_option_value (long_option, value)
            call print_version (); stop
         case ("--help")
            call no_option_value (long_option, value)
            call print_usage (); stop
         case ("--prefix")
            paths%prefix = get_option_value (i, long_option, value)
            cycle SCAN_CMDLINE
```

1301

```
case ("--exec-prefix")
   paths%exec_prefix = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--bindir")
   paths%bindir = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--libdir")
   paths%libdir = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--includedir")
   paths%includedir = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--datarootdir")
   paths%datarootdir = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--check")
   check = get_option_value (i, long_option, value)
   checks = checks // " " // check
   cycle SCAN_CMDLINE
case ("--interactive")
   call no_option_value (long_option, value)
   interactive = .true.
   cycle SCAN_CMDLINE
case ("--library")
   library = get_option_value (i, long_option, value)
   libraries = libraries // " " // library
   cycle SCAN_CMDLINE
case ("--localprefix")
   paths%localprefix = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--logfile")
   logfile = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--no-logfile")
   call no_option_value (long_option, value)
   logfile = ""
   cycle SCAN_CMDLINE
case ("--model")
   model = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--rebuild")
   call no_option_value (long_option, value)
   rebuild_library = .true.
   rebuild_phs = .true.
   rebuild_grids = .true.
   rebuild_events = .true.
   cycle SCAN_CMDLINE
case ("--rebuild-library")
   call no_option_value (long_option, value)
   rebuild_library = .true.
   cycle SCAN_CMDLINE
case ("--rebuild-phase-space")
   call no_option_value (long_option, value)
   rebuild_phs = .true.
```

```
            cycle SCAN_CMDLINE
      case ("--rebuild-grids")
         call no_option_value (long_option, value)
         rebuild_grids = .true.
         cycle SCAN_CMDLINE
      case ("--rebuild-events")
         call no_option_value (long_option, value)
         rebuild_events = .true.
         cycle SCAN_CMDLINE
      case ("--recompile")
         call no_option_value (long_option, value)
         recompile_library = .true.
         rebuild_grids = .true.
         cycle SCAN_CMDLINE
      case ("--time-estimate")
         call no_option_value (long_option, value)
         time_estimate = .true.
         cycle SCAN_CMDLINE
      case ("--no-time-estimate")
         call no_option_value (long_option, value)
         time_estimate = .false.
         cycle SCAN_CMDLINE
      case ("--write-syntax-tables")
         call no_option_value (long_option, value)
         call init_syntax_tables ()
         call write_syntax_tables ()
         call final_syntax_tables ()
         stop
         cycle SCAN_CMDLINE
      case default
         call print_usage ()
         call msg_fatal ("Option '" // trim (arg) // "' not recognized")
      end select
end select
select case (arg(1:1))
case ("-")
   j = 1
   if (len_trim (arg) == 1) then
      look_for_options = .false.
   else
      SCAN_SHORT_OPTIONS: do
         j = j + 1
         if (j > len_trim (arg)) exit SCAN_SHORT_OPTIONS
         option = "-" // arg(j:j)
         select case (option)
         case ("-V")
            call print_version (); stop
         case ("-?", "-h")
            call print_usage (); stop
         case ("-i")
            interactive = .true.
            cycle SCAN_SHORT_OPTIONS
         case ("-l")
            if (j == len_trim (arg)) then
```

```fortran
                           library = get_option_value (i, var_str (option))
                        else
                           library = trim (arg(j+1:))
                        end if
                        libraries = libraries // " " // library
                        cycle SCAN_CMDLINE
                     case ("-L")
                        if (j == len_trim (arg)) then
                           logfile = get_option_value (i, var_str (option))
                        else
                           logfile = trim (arg(j+1:))
                        end if
                        cycle SCAN_CMDLINE
                     case ("-m")
                        if (j < len_trim (arg))  call msg_fatal &
                             ("Option '" // option // "' needs a value")
                        model = get_option_value (i, var_str (option))
                        cycle SCAN_CMDLINE
                     case ("-r")
                        rebuild_library = .true.
                        rebuild_phs = .true.
                        rebuild_grids = .true.
                        rebuild_events = .true.
                        cycle SCAN_SHORT_OPTIONS
                     case default
                        call print_usage ()
                        call msg_fatal &
                             ("Option '" // option // "' not recognized")
                     end select
                  end do SCAN_SHORT_OPTIONS
            end if
      case default
         files = files // " " // trim (arg)
      end select
   else
      files = files // " " // trim (arg)
   end if
end do SCAN_CMDLINE

! Overall initialization
if (logfile /= "")  call logfile_init (logfile)
call msg_banner ()
call whizard_init &
     (preload_model=model, preload_libs=libraries, default_lib=libname, &
      rebuild_library=rebuild_library, &
      rebuild_phs=rebuild_phs, &
      rebuild_grids=rebuild_grids, &
      rebuild_events=rebuild_events, &
      recompile_library=recompile_library, &
      time_estimate=time_estimate, &
      paths=paths)

! Run any self-checks (and no commands)
if (checks /= "") then
```

```
      checks = trim (adjustl (checks))
      RUN_CHECKS: do while (checks /= "")
          call split (checks, check, " ")
          call whizard_check (check)
      end do RUN_CHECKS

  ! Process commands from standard input
  else if (.not. interactive .and. files == "") then
      call whizard_process_stdin (quit, quit_code)

  ! Process commands from file
  else
      files = trim (adjustl (files))
      SCAN_FILES: do while (files /= "")
          call split (files, this, " ")
          call whizard_process_file (this, quit, quit_code)
          if (quit)  exit SCAN_FILES
      end do SCAN_FILES

  end if

  ! Enter an interactive shell if requested
  if (.not. quit .and. interactive) then
      call whizard_shell (quit_code)
  end if

  ! Overall finalization
  call whizard_final ()
  call msg_terminate (quit_code = quit_code)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
contains

  subroutine no_option_value (option, value)
    type(string_t), intent(in) :: option, value
    if (value /= "") then
        call msg_error (" Option '" // char (option) // "' should have no value")
    end if
  end subroutine no_option_value

  function get_option_value (i, option, value) result (string)
    type(string_t) :: string
    integer, intent(inout) :: i
    type(string_t), intent(in) :: option
    type(string_t), intent(in), optional :: value
    character(CMDLINE_ARG_LEN) :: arg
    character(CMDLINE_ARG_LEN) :: arg_value
    integer :: arg_len, arg_status
    logical :: has_value
    if (present (value)) then
        has_value = value /= ""
    else
        has_value = .false.
```

```
      end if
      if (has_value) then
         string = value
      else
         i = i + 1
         call get_command_argument (i, arg_value, arg_len, arg_status)
         select case (arg_status)
         case (0)
         case (-1)
            call msg_error (" Option value truncated: '" // arg // "'")
         case default
            call print_usage ()
            call msg_fatal (" Option '" // char (option) // "' needs a value")
         end select
         select case (arg(1:1))
         case ("-")
            call print_usage ()
            call msg_fatal (" Option '" // char (option) // "' needs a value")
         end select
         string = trim (arg_value)
      end if
end function get_option_value

subroutine print_version ()
  print "(A)", "WHIZARD " // WHIZARD_VERSION
  print "(A)", "Copyright (C) 1999-2010 Wolfgang Kilian, Thorsten Ohl, Juergen Reuter"
  print "(A)", "This is free software; see the source for copying conditions.  There is NO"
  print "(A)", "warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE."
  print *
end subroutine print_version

subroutine print_usage ()
  print "(A)", "WHIZARD " // WHIZARD_VERSION
  print "(A)", "Usage: whizard [OPTIONS] [FILE]"
  print "(A)", "Run WHIZARD with the command list taken from FILE(s)"
  print "(A)", "Options for resetting default directories " &
          // "(GNU naming conventions):"
  print "(A)", "    --prefix DIR"
  print "(A)", "    --exec_prefix DIR"
  print "(A)", "    --bindir DIR"
  print "(A)", "    --libdir DIR"
  print "(A)", "    --includedir DIR"
  print "(A)", "    --datarootdir DIR"
  print "(A)", "Other options:"
  print "(A)", "-h, --help           display this help and exit"
  print "(A)", "-i, --interactive    run interactively after reading FILE(s)"
  print "(A)", "-l, --library        preload process library NAME"
  print "(A)", "    --localprefix DIR"
  print "(A)", "                      search in DIR for local models (default: ~/.whizard)"
  print "(A)", "-L, --logfile FILE   write log to FILE (default: 'whizard.log'"
  print "(A)", "-m, --model NAME     preload model NAME (default: 'SM')"
  print "(A)", "    --no-logfile      do not write a logfile"
  print "(A)", "-r, --rebuild        rebuild process code, phase space, integration grids, and
  print "(A)", "    --rebuild-library"
```

1306

```
       print "(A)", "                          rebuild process code library"
       print "(A)", "    --rebuild-phase-space"
       print "(A)", "                          rebuild phase-space configuration"
       print "(A)", "    --rebuild-grids   rebuild integration grids"
       print "(A)", "    --rebuild-events  rebuild event samples"
       print "(A)", "    --recompile       recompile process code, ignoring any existing library"
       print "(A)", "-V, --version        output version information and exit"
       print "(A)", "    --write-syntax-tables"
       print "(A)", "                          write the internal syntax tables to files and exit"
       print "(A)", "-                         further options are taken as filenames"
       print *
       print "(A)", "With no FILE, read standard input."
  end subroutine print_usage

end program main
```

Chapter 17

Cross References