GHC User's Guide Documentation

Release 9.10.0.20240313

GHC Team

Mar 13, 2024

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

This is a guide to using the Glasgow Haskell Compiler (GHC): an interactive and batch compilation system for the Haskell 2010 language.

GHC has two main components: an interactive Haskell interpreter (also known as GHCi), described in *Using GHCi* (page 15), and a batch compiler, described throughout *Using GHC* (page 65). In fact, GHC consists of a single program which is just run with different options to provide either the interactive or the batch system.

The batch compiler can be used alongside GHCi: compiled modules can be loaded into an interactive session and used in the same way as interpreted code, and in fact when using GHCi most of the library code will be pre-compiled. This means you get the best of both worlds: fast pre-compiled library code, and fast compile turnaround for the parts of your program being actively developed.

GHC supports numerous language extensions, including concurrency, a foreign function interface, exceptions, type system extensions such as multi-parameter type classes, local universal and existential quantification, functional dependencies, scoped type variables and explicit unboxed types. These are all described in *Language extensions* (page 269).

GHC has a comprehensive optimiser, so when you want to Really Go For It (and you've got time to spare) GHC can produce pretty fast code. Alternatively, the default option is to compile as fast as possible while not making too much effort to optimise the generated code (although GHC probably isn't what you'd describe as a fast compiler :-).

GHC's profiling system supports "cost centre stacks": a way of seeing the profile of a Haskell program in a call-graph like structure. See *Profiling* (page 651) for more details.

GHC comes with a number of libraries. These are described in separate documentation.

1.1 Obtaining GHC

Go to the GHC home page and follow the "download" link to download GHC for your platform.

Alternatively, if you want to build GHC yourself, head on over to the GHC Building Guide to find out how to get the sources, and build it on your system. Note that GHC itself is written in Haskell, so you will still need to install GHC in order to build it.

1.2 Meta-information: Web sites, mailing lists, etc.

On the World-Wide Web, there are several URLs of likely interest:

- GHC home page
- GHC Developers Home (developer documentation, wiki, and bug tracker)

We run the following mailing lists about GHC. We encourage you to join, as you feel is appropriate.

glasgow-haskell-users This list is for GHC users to chat among themselves. If you have a specific question about GHC, please check the FAQ first.

Subscribers can post to the list by sending their message to glasgow-haskell-users@haskell.org. Further information can be found on the Mailman page.

ghc-devs The GHC developers hang out here. If you are working with the GHC API or have a question about GHC's implementation, feel free to chime in.

Subscribers can post to the list by sending their message to ghc-devs@haskell.org. Further information can be found on the Mailman page.

There are several other Haskell and GHC-related mailing lists served by `www.haskell.org`. Go to https://www.haskell.org/mailman/listinfo/ for the full list.

1.3 Reporting bugs in GHC

Glasgow Haskell is a changing system so there are sure to be bugs in it. If you find one, please see this wiki page for information on how to report it.

1.4 GHC version numbering policy

As of GHC version 6.8, we have adopted the following policy for numbering GHC versions:

Stable branches are numbered `x.y`, where ⟨y⟩ is *even*. Releases on the stable branch `x.y` are numbered `x.y.z`, where ⟨z⟩ (>= 1) is the patchlevel number. Patchlevels are bug-fix releases only, and never change the programmer interface to any system-supplied code. However, if you install a new patchlevel over an old one you will need to recompile any code that was compiled against the old libraries.

The value of `__GLASGOW_HASKELL__` (see *Options affecting the C pre-processor* (page 240)) for a major release `x.y.z` is the integer ⟨xyy⟩ (if ⟨y⟩ is a single digit, then a leading zero is added, so for example in version 6.8.2 of GHC we would have `__GLASGOW_HASKELL__==608`).

We may make snapshot releases of the current stable branch available for download, and the latest sources are available from the git repositories.

Stable snapshot releases are named `x.y.z.YYYYMMDD`. where `YYYYMMDD` is the date of the sources from which the snapshot was built, and `x.y.z+1` is the next release to be made on that branch. For example, `6.8.1.20040225` would be a snapshot of the `6.8` branch during the development of `6.8.2`.

> Unstable snapshot releases are named `x.y.YYYYMMDD`. where `YYYYMMDD` is the date of the sources from which the snapshot was built. For example, `6.7.20040225` would be a snapshot of the HEAD before the creation of the `6.8` branch.
>
> The value of `__GLASGOW_HASKELL__` for a snapshot release is the integer ⟨xyy⟩. You should never write any conditional code which tests for this value, however: since interfaces change on a day-to-day basis, and we don't have finer granularity in the values of `__GLASGOW_HASKELL__`, you should only conditionally compile using predicates which test whether `__GLASGOW_HASKELL__` is equal to, later than, or earlier than a given major release.

The version number of your copy of GHC can be found by invoking `ghc` with the `--version` flag (see *Verbosity options* (page 76)).

The compiler version can be tested within compiled code with the `MIN_VERSION_GLASGOW_HASKELL` CPP macro (defined only when *CPP* (page 240) is used). See *Standard CPP macros* (page 241) for details.

1.5 The Glasgow Haskell Compiler License

Copyright 2002 - 2007, The University Court of the University of Glasgow. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY COURT OF THE UNIVERSITY OF GLASGOW AND THE CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE UNIVERSITY COURT OF THE UNIVERSITY OF GLASGOW OR THE CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CHAPTER TWO

RELEASE NOTES

2.1 Version 9.10.1

2.1.1 Language

- The *GHC2024* (page 270) language edition is now supported. It builds on top of *GHC2021* (page 271), adding the following extensions:
  - *DataKinds* (page 357)
  - *DerivingStrategies* (page 455)
  - *DisambiguateRecordFields* (page 423)
  - *ExplicitNamespaces* (page 319)
  - *GADTs* (page 335)
  - *MonoLocalBinds* (page 526)
  - *LambdaCase* (page 299)
  - *RoleAnnotations* (page 419)

  At the moment, *GHC2021* (page 271) remains the default langauge edition that is used when no other language edition is explicitly loaded (e.g. when running `ghc` directly). Because language editions are not necessarily backwards compatible, and future releases of GHC may change the default, it is highly recommended to specify the language edition explicitly.

- GHC Proposal #281 "Visible forall in types of terms" has been partially implemented. The following code is now accepted by GHC:

  ```
  {-# LANGUAGE RequiredTypeArguments #-}

  vshow :: forall a -> Show a => a -> String
  vshow t x = show (x :: t)

  s1 = vshow Int    42      -- "42"
  s2 = vshow Double 42      -- "42.0"
  ```

  The use of `forall a ->` instead of `forall a.` indicates a *required* type argument. A required type argument is visually indistinguishable from a value argument but does not exist at runtime.

  This feature is guarded behind *RequiredTypeArguments* (page 395).

- The *ExplicitNamespaces* (page 319) extension can now be used in conjunction with *RequiredTypeArguments* (page 395) to select the type namespace in a required type argument:

```
data T = T               -- the name `T` is ambiguous
f :: forall a -> ...     -- `f` expects a required type argument

x1 = f T         -- refers to the /data/ constructor `T`
x2 = f (type T)  -- refers to the /type/ constructor `T`
```

- With *LinearTypes* (page 408), `let` and `where` bindings can now be linear. So the following now typechecks:

```
f :: A %1 -> B
g :: B %1 -> C

h :: A %1 -> C
h x = g y
  where
    y = f x
```

- Due to an oversight, previous GHC releases (starting from 9.4) allowed the use of promoted data types in kinds, even when *DataKinds* (page 357) was not enabled. That is, GHC would erroneously accept the following code:

```
{-# LANGUAGE NoDataKinds #-}

import Data.Kind (Type)
import GHC.TypeNats (Nat)

-- Nat shouldn't be allowed here without DataKinds
data Vec :: Nat -> Type -> Type
```

  This oversight has now been fixed. If you wrote code that took advantage of this oversight, you may need to enable *DataKinds* (page 357) in your code to allow it to compile with GHC 9.10.

  For more information on what types are allowed in kinds, see the *Datatype promotion* (page 357) section.

- Using `forall` as an identifier is now a parse error, as forewarned by *-Wforall-identifier* (page 108):

```
forall :: (Variable a, MonadQSAT s m) => m a
-- parse error on input ‘forall’
```

  Library authors are advised to use a different name for their functions, such as `forAll`, `for_all`, or `forall_`.

- GHC Proposal #65 “Require namespacing fixity declarations for type names and WARNING/DEPRECATED pragmas” has been partially implemented. Now, with *ExplicitNamespaces* (page 319) enabled, you can specify the namespace of a name in fixity signatures, `DEPRECATED` and `WARNING` pragmas:

```
type f $ a = f a
f $ a = f a

infixl 9 type $ -- type-level $ is left-associative with priority 9
```

(continues on next page)

(continued from previous page)

```
infixr 0 data $ -- term-level $ is right-associative with priority 0

{-# DEPRECATED type D "Use `()` instead" #-} -- this will deprecate type D, but
↪will not touch pattern synonym
data D = MkD

{-# DEPRECATED data D "Use `MkD` instead" #-} -- this will deprecate pattern
↪synonym only
pattern D = MkD

pattern Head x <- (head -> x)
{-# WARNING in "x-partial" data Head [ "This is a partial synonym,"
                                     , "it throws an error on empty lists."] #-}
```

- GHC Proposal #475 "Non-punning list and tuple syntax" has been partially implemented. When the newly introduced extension *ListTuplePuns* (page 361) is disabled, bracket syntax for lists, tuples and sums only denotes their data constructors, while their type constructors have been changed to use regular prefix syntax:

  ```
  data List a = [] | a : List a
  data Tuple2 a b = (a, b)
  ```

  The extension is enabled by default, establishing the usual behavior.

- In accordance with GHC Proposal #448, the *TypeAbstractions* (page 390) extension has been extended to support @-binders in lambdas and function equations:

  ```
  id :: forall a. a -> a
  id @t x = x :: t
  -- ^^ @-binder in a function equation

  e = higherRank (\ @t -> ... )
  --                ^^ @-binder in a lambda
  ```

  This feature is an experimental alternative to *ScopedTypeVariables* (page 512), see the *Type Abstractions in Functions* (page 392) section.

2.1.2 Compiler

- GHC Proposal #516 has been implemented. It introduces a warning *-Wincomplete-record-selectors* (page 95) which warns about when an invocation of a record selector may fail due to being applied to a constructor for which it is not defined.

  For example

  ```
  data T = T1 | T2 { x :: Int }
  f :: T -> Int
  f a = x a + 1 -- emit a warning here, since `f T1` will fail
  ```

  Unlike *-Wpartial-fields* (page 106) this produces a warning about incomplete selectors at use sites instead of definition sites, so it is useful in cases when the library does intend for incomplete record selectors to be used but only in specific circumstances (e.g. when other cases are handled by previous pattern matches).

- The `-finfo-table-map-with-stack` (page 687) and `-finfo-table-map-with-fallback` (page 687) flags have been introduced. These flags include `STACK` info tables and info tables with default source location information in the info table map, respectively. They are implied by the `-finfo-table-map` (page 687) flag. The corresponding negative flags (`-fno-info-table-map-with-stack` (page 687), `-fno-info-table-map-with-fallback` (page 687)) are useful for omitting these info tables from the info table map and reducing the size of executables containing info table profiling information. In a test on the Agda codebase, the size of the build results was reduced by about 10% when these info tables were omitted.
- Fixed a bug where compiling with both `-ddump-timings` (page 256) and `-ddump-to-file` (page 255) did not suppress printing timings to the console. See #20316.
- Defaulting plugins can now propose solutions to entangled sets of type variables. This allows defaulting of multi-parameter type classes. See #23832.
- The flag *-funbox-small-strict-fields* will now properly recognize unboxed tuples containing multiple elements as large. Constructors like *Foo (# Int64, Int64#)* will no longer be considered small and therefore not unboxed by default under *-O* even when used as strict field. #22309.
- The flag *-funbox-small-strict-fields* will now always unpack things as if compiling for a 64bit platform. Even when generating code for a 32bit platform. This makes core optimizations more consistent between 32bit and 64bit platforms at the cost of slightly worse 32bit performance in edge cases.
- Type abstractions in constructor patterns that were previously admitted without enabling the `TypeAbstractions` (page 390) extension now trigger a warning, `-Wdeprecated-type-abstractions` (page 110). This new warning is part of the `-Wcompat` (page 86) warning group and will become an error in a future GHC release.
- The `-Wforall-identifier` (page 108) flag is now deprecated and removed from `-Wdefault` (page 85), as `forall` is no longer parsed as an identifier.
- Late plugins have been added. These are plugins which can access and/or modify the core of a module after optimization and after interface creation. See #24254.
- If you use `-fllvm` (page 243) we now use an assembler from the LLVM toolchain rather than the preconfigured assembler. This is typically `clang`. The `LLVMAS` environment variable can be specified at configure time to instruct GHC which `clang` to use. This means that if you are using `-fllvm` you now need `llc`, `opt` and `clang` available.
- The `-fprof-late-overloaded` (page 658) flag has been introduced. It causes cost centres to be added to *overloaded* top level bindings, unlike `-fprof-late` (page 657) which adds cost centres to all top level bindings.
- The `-fprof-late-overloaded-calls` (page 658) flag has been introduced. It causes cost centres to be inserted at call sites including instance dictionary arguments. This may be preferred over `-fprof-late-overloaded` (page 658) since it may reveal whether imported functions are called overloaded.

2.1.3 JavaScript backend

- The JavaScript backend now supports linking with C sources. It uses Emscripten to compile them to WebAssembly. The resulting JS file embeds and loads these WebAssembly files. Important note: JavaScript wrappers are required to call into C functions and pragmas have been added to indicate which C functions are exported (see the users guide).

2.1.4 WebAssembly backend

- The wasm backend now implements JavaScript FFI, allowing JavaScript to be called from Haskell and vice versa when targetting JavaScript environments like browsers and node.js. See *JavaScript FFI in the wasm backend* (page 722) for details.

2.1.5 GHCi

- GHCi now differentiates between adding, unadding, loading, unloading and reloading in its responses to using the respective commands. The output with *-fshow-loaded-modules* is not changed to keep backwards compatibility for tooling.

2.1.6 Runtime system

- Internal fragmentation incurred by the non-moving GC's allocator has been reduced for small objects. In one real-world application, this has reduced resident set size by about 20% and modestly improved run-time. See #23340. *--nonmoving-dense-allocator-count=⟨count⟩* (page 184) has been added to fine-tune this behaviour.
- Add support for heap profiling with the non-moving GC. See #22221.
- Add a *--no-automatic-time-samples* (page 667) flag which stops time profiling samples being automatically started on startup. Time profiling can be controlled manually using functions in `GHC.Profiling`.
- Add a *-xr ⟨size⟩* (page 183) which controls the size of virtual memory address space reserved by the two step allocator on a 64-bit platform. The default size is now 1T on aarch64 as well. See #24498.

2.1.7 base library

- Updated to Unicode 15.1.0.
- The functions GHC.Exts.dataToTag# and GHC.Base.getTag have had their types changed to the following:

  ```
  dataToTag#, getTag
    :: forall {lev :: Levity} (a :: TYPE (BoxedRep lev))
    .  DataToTag a => a -> Int#
  ```

  In particular, they are now applicable only at some (not all) lifted types. However, if `t` is an algebraic data type (i.e. `t` matches a `data` or `data instance` declaration) with all of its constructors in scope and the levity of `t` is statically known, then the constraint `DataToTag t` can always be solved.

- Exceptions can now carry arbitrary user-defined annotations via the new GHC.Exception.Type.ExceptionContext implicit parameter of `SomeException`. These annotations are intended to be used to carry context describing the provenance of an exception.
- GHC now collects backtraces for synchronous exceptions. These are carried by the exception via the `ExceptionContext` mechanism described above. GHC supports several mechanisms by which backtraces can be collected which can be individually enabled and disabled via GHC.Exception.Backtrace.setEnabledBacktraceMechanisms.

2.1.8 `ghc-prim` library

- `dataToTag#` has been moved from `GHC.Prim`. It remains exported by `GHC.Exts`, but with a different type, as described in the notes for `base` above.
- New primops for unaligned `Addr#` access. These primops will be emulated on platforms that don't support unaligned access. These primops take the form

```
indexWord8OffAddrAs<ty> :: Addr# -> Int# -> <ty>#
readWord8OffAddrAs<ty> :: Addr# -> Int# -> State# s -> (# State# s, <ty># #)
writeWord8OffAddrAs<ty> :: Addr# -> Int# -> <ty># -> State# s -> State# s
```

  where `<ty>` is one of:

  - `Word`
  - `Word{16,32,64}`
  - `Int`
  - `Int{16,32,64,}`
  - `Char`
  - `WideChar`
  - `Addr`
  - `Float`
  - `Double`
  - `StablePtr`

2.1.9 `ghc` library

2.1.10 `ghc-heap` library

2.1.11 `ghc-experimental` library

- `ghc-experimental` is a new library for functions and data types with weaker stability guarantees. Introduced per the HF Technical Proposal #51.

2.1.12 `template-haskell` library

- Extend `Pat` with `TypeP` and `Exp` with `TypeE`, introduce functions `typeP` and `typeE` (Template Haskell support for GHC Proposal #281).

2.1.13 Included libraries

The package database provided with this distribution also contains a number of packages other than GHC itself. See the changelogs provided with these packages for further change information.

Package	Version	Reason for inclusion
ghc	9.10.0.20240313	The compiler itself
Cabal-syntax	3.11.0.0	Dependency of `ghc-pkg` utility
Cabal	3.11.0.0	Dependency of `ghc-pkg` utility
Win32	2.14.0.0	Dependency of `ghc` library
array	0.5.6.0	Dependency of `ghc` library
base	4.20.0.0	Core library
binary	0.8.9.1	Dependency of `ghc` library
bytestring	0.12.1.0	Dependency of `ghc` library
containers	0.7	Dependency of `ghc` library
deepseq	1.5.0.0	Dependency of `ghc` library
directory	1.3.8.3	Dependency of `ghc` library
exceptions	0.10.7	Dependency of `ghc` and `haskeline` library
filepath	1.5.2.0	Dependency of `ghc` library
ghc-boot-th	9.10.0.20240313	Internal compiler library
ghc-boot	9.10.0.20240313	Internal compiler library
ghc-compact	0.1.0.0	Core library
ghc-heap	9.10.0.20240313	GHC heap-walking library
ghc-prim	0.11.0	Core library

Continued on next page

Table 1 – continued from previous page

Package	Version	Reason for inclusion
ghci	9.10.0.20240313	The REPL interface
haskeline	0.8.2.1	Dependency of `ghci` executable
hpc	0.7.0.1	Dependency of `hpc` executable
integer-gmp	1.1	Core library
mtl	2.3.1	Dependency of `Cabal` library
parsec	3.1.16.1	Dependency of `Cabal` library
pretty	1.1.3.6	Dependency of `ghc` library
process	1.6.18.0	Dependency of `ghc` library
stm	2.5.3.0	Dependency of `haskeline` library
template-haskell	2.22.0.0	Core library
terminfo	0.4.1.6	Dependency of `haskeline` library
text	2.1.1	Dependency of `Cabal` library
time	1.12.2	Dependency of `ghc` library
transformers	0.6.1.1	Dependency of `ghc` library
unix	2.8.5.0	Dependency of `ghc` library
xhtml	3000.2.2.1	Dependency of `haddock` executable

CHAPTER THREE

USING GHCI

GHCi[1] is GHC's interactive environment that includes an interactive debugger (see *The GHCi Debugger* (page 32)).

GHCi can

- interactively evaluate Haskell expressions
- interpret Haskell programs
- load GHC-compiled modules.

At the moment GHCi supports most of GHC's language extensions.

3.1 Introduction to GHCi

Let's start with an example GHCi session. You can fire up GHCi with the command `ghci`:

```
$ ghci
GHCi, version 8.y.z: https://www.haskell.org/ghc/  :? for help
ghci>
```

There may be a short pause while GHCi loads the prelude and standard libraries, after which the prompt is shown. As the banner says, you can type *:?* (page 48) to see the list of commands available, and a half line description of each of them. We'll explain most of these commands as we go along, and there is complete documentation for all the commands in *GHCi commands* (page 44).

Haskell expressions can be typed at the prompt:

```
ghci> 1+2
3
ghci> let x = 42 in x / 9
4.666666666666667
ghci>
```

GHCi interprets the whole line as an expression to evaluate. The expression may not span several lines - as soon as you press enter, GHCi will attempt to evaluate it.

In Haskell, a `let` expression is followed by `in`. However, in GHCi, since the expression can also be interpreted in the `IO` monad, a `let` binding with no accompanying `in` statement can be signalled by an empty line, as in the above example.

Since GHC 8.0.1, you can bind values and functions to names without `let` statement:

[1] The "i" stands for "Interactive"

```
ghci> x = 42
ghci> x
42
ghci>
```

3.2 Loading source files

Suppose we have the following Haskell source code, which we place in a file `Main.hs`:

```
main = print (fac 20)

fac 0 = 1
fac n = n * fac (n-1)
```

You can save `Main.hs` anywhere you like, but if you save it somewhere other than the current directory[3] then we will need to change to the right directory in GHCi:

```
ghci> :cd dir
```

where ⟨dir⟩ is the directory (or folder) in which you saved `Main.hs`.

To load a Haskell source file into GHCi, use the *:load* (page 50) command:

```
ghci> :load Main
Compiling Main             ( Main.hs, interpreted )
Ok, modules loaded: Main.
*ghci>
```

GHCi has loaded the `Main` module, and the prompt has changed to `*ghci>` to indicate that the current context for expressions typed at the prompt is the `Main` module we just loaded (we'll explain what the * means later in *What's really in scope at the prompt?* (page 24)). So we can now type expressions involving the functions from `Main.hs`:

```
*ghci> fac 17
355687428096000
```

Loading a multi-module program is just as straightforward; just give the name of the "topmost" module to the *:load* (page 50) command (hint: *:load* (page 50) can be abbreviated to `:l`). The topmost module will normally be `Main`, but it doesn't have to be. GHCi will discover which modules are required, directly or indirectly, by the topmost module, and load them all in dependency order.

-fshow-loaded-modules

> **Default** off
>
> **Since** 8.2.2
>
> Typically GHCi will show only the number of modules that it loaded after a *:load* (page 50) command. With this flag, GHC will also list the loaded modules' names. This was the default behavior prior to GHC 8.2.1 and can be useful for some tooling users.

[3] If you started up GHCi from the command line then GHCi's current directory is the same as the current directory of the shell from which it was started. If you started GHCi from the "Start" menu in Windows, then the current directory is probably something like `C:\Documents and Settings\user name`.

3.2.1 Modules vs. filenames

Question: How does GHC find the filename which contains module ⟨M⟩? Answer: it looks for the file `M.hs`, or `M.lhs`. This means that for most modules, the module name must match the filename. If it doesn't, GHCi won't be able to find it.

There is one exception to this general rule: when you load a program with *:load* (page 50), or specify it when you invoke `ghci`, you can give a filename rather than a module name. This filename is loaded if it exists, and it may contain any module you like. This is particularly convenient if you have several `Main` modules in the same directory and you can't call them all `Main.hs`.

The search path for finding source files is specified with the *-i* (page 201) option on the GHCi command line, like so:

```
ghci -idir1:...:dirn
```

or it can be set using the *:set* (page 52) command from within GHCi (see *Setting GHC command-line options in GHCi* (page 57))[4]

One consequence of the way that GHCi follows dependencies to find modules to load is that every module must have a source file. The only exception to the rule is modules that come from a package, including the `Prelude` and standard libraries such as `IO` and `Complex`. If you attempt to load a module for which GHCi can't find a source file, even if there are object and interface files for the module, you'll get an error message.

3.2.2 Making changes and recompilation

If you make some changes to the source code and want GHCi to recompile the program, give the *:reload* (page 51) command. The program will be recompiled as necessary, with GHCi doing its best to avoid actually recompiling modules if their external dependencies haven't changed. This is the same mechanism we use to avoid re-compiling modules in the batch compilation setting (see *The recompilation checker* (page 206)).

3.3 Loading compiled code

When you load a Haskell source module into GHCi, it is normally converted to byte-code and run using the interpreter. However, interpreted code can also run alongside compiled code in GHCi; indeed, normally when GHCi starts, it loads up a compiled copy of the `base` package, which contains the `Prelude`.

Why should we want to run compiled code? Well, compiled code is roughly 10x faster than interpreted code, but takes about 2x longer to produce (perhaps longer if optimisation is on). So it pays to compile the parts of a program that aren't changing very often, and use the interpreter for the code being actively developed.

When loading up source modules with *:load* (page 50), GHCi normally looks for any corresponding compiled object files, and will use one in preference to interpreting the source if possible. For example, suppose we have a 4-module program consisting of modules `A`, `B`, `C`, and `D`. Modules `B` and `C` both import `D` only, and `A` imports both `B` and `C`:

[4] Note that in GHCi, and *--make* (page 68) mode, the *-i* (page 201) option is used to specify the search path for *source* files, whereas in standard batch-compilation mode the *-i* (page 201) option is used to specify the search path for interface files, see *The search path* (page 201).

```
  A
 / \
B   C
 \ /
  D
```

We can compile D, then load the whole program, like this:

```
ghci> :! ghc -c -dynamic D.hs
ghci> :load A
Compiling B                ( B.hs, interpreted )
Compiling C                ( C.hs, interpreted )
Compiling A                ( A.hs, interpreted )
Ok, modules loaded: A, B, C, D (D.o).
*ghci>
```

In the messages from the compiler, we see that there is no line for D. This is because it isn't necessary to compile D, because the source and everything it depends on is unchanged since the last compilation.

Note the *-dynamic* (page 246) flag to GHC: GHCi uses dynamically-linked object code (if you are on a platform that supports it), and so in order to use compiled code with GHCi it must be compiled for dynamic linking.

At any time you can use the command *:show modules* (page 54) to get a list of the modules currently loaded into GHCi:

```
*ghci> :show modules
D                ( D.hs, D.o )
C                ( C.hs, interpreted )
B                ( B.hs, interpreted )
A                ( A.hs, interpreted )
*ghci>
```

If we now modify the source of D (or pretend to: using the Unix command touch on the source file is handy for this), the compiler will no longer be able to use the object file, because it might be out of date:

```
*ghci> :! touch D.hs
*ghci> :reload
Compiling D                ( D.hs, interpreted )
Ok, modules loaded: A, B, C, D.
*ghci>
```

Note that module D was compiled, but in this instance because its source hadn't really changed, its interface remained the same, and the recompilation checker determined that A, B and C didn't need to be recompiled.

So let's try compiling one of the other modules:

```
*ghci> :! ghc -c C.hs
*ghci> :load A
Compiling D                ( D.hs, interpreted )
Compiling B                ( B.hs, interpreted )
Compiling C                ( C.hs, interpreted )
Compiling A                ( A.hs, interpreted )
Ok, modules loaded: A, B, C, D.
```

We didn't get the compiled version of `C`! What happened? Well, in GHCi a compiled module may only depend on other compiled modules, and in this case `C` depends on `D`, which doesn't have an object file, so GHCi also rejected `C`'s object file. Ok, so let's also compile `D`:

```
*ghci> :! ghc -c D.hs
*ghci> :reload
Ok, modules loaded: A, B, C, D.
```

Nothing happened! Here's another lesson: newly compiled modules aren't picked up by *:reload* (page 51), only *:load* (page 50):

```
*ghci> :load A
Compiling B                ( B.hs, interpreted )
Compiling A                ( A.hs, interpreted )
Ok, modules loaded: A, B, C (C.o), D (D.o).
```

The automatic loading of object files can sometimes lead to confusion, because non-exported top-level definitions of a module are only available for use in expressions at the prompt when the module is interpreted (see *What's really in scope at the prompt?* (page 24)). For this reason, you might sometimes want to force GHCi to load a module using the interpreter. This can be done by prefixing a * to the module name or filename when using *:load* (page 50), for example

```
ghci> :load *A
Compiling A                ( A.hs, interpreted )
*ghci>
```

When the * is used, GHCi ignores any pre-compiled object code and interprets the module. If you have already loaded a number of modules as object code and decide that you wanted to interpret one of them, instead of re-loading the whole set you can use `:add *M` to specify that you want `M` to be interpreted (note that this might cause other modules to be interpreted too, because compiled modules cannot depend on interpreted ones).

To always compile everything to object code and never use the interpreter, use the *-fobject-code* (page 244) option (see *Compiling to object code inside GHCi* (page 60)).

Hint: Since GHCi will only use a compiled object file if it can be sure that the compiled version is up-to-date, a good technique when working on a large program is to occasionally run `ghc --make` to compile the whole project (say before you go for lunch :-), then continue working in the interpreter. As you modify code, the changed modules will be interpreted, but the rest of the project will remain compiled.

3.4 Interactive evaluation at the prompt

When you type an expression at the prompt, GHCi immediately evaluates and prints the result:

```
ghci> reverse "hello"
"olleh"
ghci> 5+5
10
```

3.4.1 I/O actions at the prompt

GHCi does more than simple expression evaluation at the prompt. If you enter an expression of type `IO a` for some `a`, then GHCi *executes* it as an IO-computation.

```
ghci> "hello"
"hello"
ghci> putStrLn "hello"
hello
```

This works even if the type of the expression is more general, provided it can be *instantiated* to `IO a`. For example

```
ghci> return True
True
```

Furthermore, GHCi will print the result of the I/O action if (and only if):

- The result type is an instance of `Show`.
- The result type is not `()`.

For example, remembering that `putStrLn :: String -> IO ()`:

```
ghci> putStrLn "hello"
hello
ghci> do { putStrLn "hello"; return "yes" }
hello
"yes"
```

3.4.2 Using `do` notation at the prompt

GHCi actually accepts statements rather than just expressions at the prompt. This means you can bind values and functions to names, and use them in future expressions or statements.

The syntax of a statement accepted at the GHCi prompt is exactly the same as the syntax of a statement in a Haskell `do` expression. However, there's no monad overloading here: statements typed at the prompt must be in the `IO` monad.

```
ghci> x <- return 42
ghci> print x
42
ghci>
```

The statement `x <- return 42` means "execute `return 42` in the `IO` monad, and bind the result to `x`". We can then use `x` in future statements, for example to print it as we did above.

`-fprint-bind-result`

If *`-fprint-bind-result`* (page 20) is set then GHCi will print the result of a statement if and only if:

- The statement is not a binding, or it is a monadic binding (`p <- e`) that binds exactly one variable.
- The variable's type is not polymorphic, is not `()`, and is an instance of `Show`.

Of course, you can also bind normal non-IO expressions using the `let`-statement:

```
ghci> let x = 42
ghci> x
42
ghci>
```

Another important difference between the two types of binding is that the monadic bind (`p <- e`) is *strict* (it evaluates `e`), whereas with the `let` form, the expression isn't evaluated immediately:

```
ghci> let x = error "help!"
ghci> print x
*** Exception: help!
ghci>
```

Note that `let` bindings do not automatically print the value bound, unlike monadic bindings.

You can also define functions at the prompt:

```
ghci> add a b = a + b
ghci> add 1 2
3
ghci>
```

However, this quickly gets tedious when defining functions with multiple clauses, or groups of mutually recursive functions, because the complete definition has to be given on a single line, using explicit semicolons instead of layout:

```
ghci> f op n [] = n ; f op n (h:t) = h `op` f op n t
ghci> f (+) 0 [1..3]
6
ghci>
```

:{
:}

Begin or end a multi-line GHCi command block.

To alleviate this issue, GHCi commands can be split over multiple lines, by wrapping them in `:{` and `:}` (each on a single line of its own):

```
ghci> :{
ghci| g op n [] = n
ghci| g op n (h:t) = h `op` g op n t
ghci| :}
ghci> g (*) 1 [1..3]
6
```

Such multiline commands can be used with any GHCi command, and note that the layout rule is in effect. The main purpose of multiline commands is not to replace module loading but to make definitions in .ghci-files (see *The .ghci and .haskeline files* (page 58)) more readable and maintainable.

Any exceptions raised during the evaluation or execution of the statement are caught and printed by the GHCi command line interface (for more information on exceptions, see the module Control.Exception in the libraries documentation.)

Every new binding shadows any existing bindings of the same name, including entities that are in scope in the current module context.

Warning: Temporary bindings introduced at the prompt only last until the next *:load* (page 50), *:reload* (page 51), *:add* (page 44) or *:unadd* (page 55) command, at which time they will be simply lost. However, they do survive a change of context with *:module* (page 51): the temporary bindings just move to the new location.

Hint: To get a list of the bindings currently in scope, use the *:show bindings* (page 53) command:

```
ghci> :show bindings
x :: Int
ghci>
```

Hint: If you turn on the `+t` option, GHCi will show the type of each variable bound by a statement. For example:

```
ghci> :set +t
ghci> let (x:xs) = [1..]
x :: Integer
xs :: [Integer]
```

3.4.3 Multiline input

Apart from the `:{ ... :}` syntax for multi-line input mentioned above, GHCi also has a multiline mode, enabled by `:set +m`, `:set +m` in which GHCi detects automatically when the current statement is unfinished and allows further lines to be added. A multi-line input is terminated with an empty line. For example:

```
ghci> :set +m
ghci> let x = 42
ghci|
```

Further bindings can be added to this `let` statement, so GHCi indicates that the next line continues the previous one by changing the prompt. Note that layout is in effect, so to add more bindings to this `let` we have to line them up:

```
ghci> :set +m
ghci> let x = 42
ghci|     y = 3
ghci|
ghci>
```

Explicit braces and semicolons can be used instead of layout:

```
ghci> do {
ghci| putStrLn "hello"
ghci| ;putStrLn "world"
ghci| }
hello
world
ghci>
```

Note that after the closing brace, GHCi knows that the current statement is finished, so no empty line is required.

Multiline mode is useful when entering monadic do statements:

```
ghci> flip evalStateT 0 $ do
ghci| i <- get
ghci| lift $ do
ghci|   putStrLn "Hello World!"
ghci|   print i
ghci|
"Hello World!"
0
ghci>
```

During a multiline interaction, the user can interrupt and return to the top-level prompt.

```
ghci> do
ghci| putStrLn "Hello, World!"
ghci| ^C
ghci>
```

3.4.4 Type, class and other declarations

At the GHCi prompt you can also enter any top-level Haskell declaration, including `data`, `type`, `newtype`, `class`, `instance`, `deriving`, and `foreign` declarations. For example:

```
ghci> data T = A | B | C deriving (Eq, Ord, Show, Enum)
ghci> [A ..]
[A,B,C]
ghci> :i T
data T = A | B | C      -- Defined at <interactive>:2:6
instance Enum T -- Defined at <interactive>:2:45
instance Eq T -- Defined at <interactive>:2:30
instance Ord T -- Defined at <interactive>:2:34
instance Show T -- Defined at <interactive>:2:39
```

As with ordinary variable bindings, later definitions shadow earlier ones, so you can re-enter a declaration to fix a problem with it or extend it. But there's a gotcha: when a new type declaration shadows an older one, there might be other declarations that refer to the old type. The thing to remember is that the old type still exists, and these other declarations still refer to the old type. However, while the old and the new type have the same name, GHCi will treat them as distinct. For example:

```
ghci> data T = A | B
ghci> let f A = True; f B = False
ghci> data T = A | B | C
ghci> f A

<interactive>:2:3:
    Couldn't match expected type `main::Interactive.T'
                with actual type `T'
    In the first argument of `f', namely `A'
    In the expression: f A
    In an equation for `it': it = f A
ghci>
```

The old, shadowed, version of `T` is displayed as `main::Interactive.T` by GHCi in an attempt to distinguish it from the new `T`, which is displayed as simply `T`.

Class and type-family instance declarations are simply added to the list of available instances, with one exception. Since you might want to re-define one, a class instance *replaces* any earlier instance with an identical head. You aren't allowed to re-define a type family instance, since it might not be type safe to do so. Instead, re-define the whole type-family. (See *Type families* (page 338).) For example:

```
ghci> type family T a b
ghci> type instance T a b = a
ghci> let uc :: a -> T a b; uc = id

ghci> type instance T a b = b

<interactive>:3:15: error:
    Conflicting family instance declarations:
      T a b = a -- Defined at <interactive>:3:15
      T a b = b -- Defined at <interactive>:5:15

-- Darn! We have to re-declare T.

ghci> type family T a b
-- This is a brand-new T, unrelated to the old one
ghci> type instance T a b = b
ghci> uc 'a' :: Int

<interactive>:8:1: error:
    • Couldn't match type ‘Char’ with ‘Int’
      Expected type: Int
        Actual type: Ghci1.T Char b0
    • In the expression: uc 'a' :: Int
      In an equation for ‘it’: it = uc 'a' :: Int
```

3.4.5 What's really in scope at the prompt?

When you type an expression at the prompt, what identifiers and types are in scope? GHCi provides a flexible way to control exactly how the context for an expression is constructed:

- The *:load* (page 50), *:reload* (page 51), *:add* (page 44) and *:unadd* (page 55) commands (*The effect of :load on what is in scope* (page 25)).
- The `import` declaration (*Controlling what is in scope with import* (page 25)).
- The *:module* (page 51) command (*Controlling what is in scope with the :module command* (page 26)).

The command *:show imports* (page 53) will show a summary of which modules contribute to the top-level scope.

Hint: GHCi will tab-complete names that are in scope; for example, if you run GHCi and type `J<tab>` then GHCi will expand it to `Just`.

3.4.5.1 The effect of `:load` on what is in scope

The *:load* (page 50), *:reload* (page 51), *:add* (page 44) and *:unadd* (page 55) commands (*Loading source files* (page 16) and *Loading compiled code* (page 17)) affect the top-level scope. Let's start with the simple cases; when you start GHCi the prompt looks like this:

```
ghci>
```

By default, this means that everything from the module `Prelude` is currently in scope. Should the prompt be set to `%s>` in the `.ghci` configuration file, we would be seeing `Prelude>` displayed. However, it is not the default mechanism due to the large space the prompt can take if more imports are done.

The syntax in the prompt `*module` indicates that it is the full top-level scope of ⟨module⟩ that is contributing to the scope for expressions typed at the prompt. Without the `*`, just the exports of the module are visible.

Note: For technical reasons, GHCi can only support the `*`-form for modules that are interpreted. Compiled modules and package modules can only contribute their exports to the current scope. To ensure that GHCi loads the interpreted version of a module, add the `*` when loading the module, e.g. `:load *M`.

In general, after a *:load* (page 50) command, an automatic import is added to the scope for the most recently loaded "target" module, in a `*`-form if possible. For example, if you say `:load foo.hs bar.hs` and `bar.hs` contains module `Bar`, then the scope will be set to `*Bar` if `Bar` is interpreted, or if `Bar` is compiled it will be set to `Prelude` and `Bar` (GHCi automatically adds `Prelude` if it isn't present and there aren't any `*`-form modules). These automatically-added imports can be seen with *:show imports* (page 53):

```
ghci> :load hello.hs
[1 of 1] Compiling Main             ( hello.hs, interpreted )
Ok, modules loaded: Main.
*ghci> :show imports
:module +*Main -- added automatically
*ghci>
```

and the automatically-added import is replaced the next time you use *:load* (page 50), *:reload* (page 51), *:add* (page 44) or *:unadd* (page 55). It can also be removed by *:module* (page 51) as with normal imports.

3.4.5.2 Controlling what is in scope with `import`

We are not limited to a single module: GHCi can combine scopes from multiple modules, in any mixture of `*` and non-`*` forms. GHCi combines the scopes from all of these modules to form the scope that is in effect at the prompt.

To add modules to the scope, use ordinary Haskell `import` syntax:

```
ghci> import System.IO
ghci> hPutStrLn stdout "hello\n"
hello
```

The full Haskell import syntax is supported, including `hiding` and `as` clauses. The prompt shows the modules that are currently imported, but it omits details about `hiding`, `as`, and so on. To see the full story, use *:show imports* (page 53):

```
ghci> import System.IO
ghci> import Data.Map as Map
ghci Map> :show imports
import Prelude -- implicit
import System.IO
import Data.Map as Map
```

Note that the `Prelude` import is marked as implicit. It can be overridden with an explicit `Prelude` import, just like in a Haskell module.

With multiple modules in scope, especially multiple *-form modules, it is likely that name clashes will occur. Haskell specifies that name clashes are only reported when an ambiguous identifier is used, and GHCi behaves in the same way for expressions typed at the prompt.

3.4.5.3 Controlling what is in scope with the `:module` command

Another way to manipulate the scope is to use the *:module* (page 51) command, whose syntax is this:

```
:module +|- *mod1 ... *modn
```

Using the `+` form of the `module` commands adds modules to the current scope, and `-` removes them. Without either `+` or `-`, the current scope is replaced by the set of modules specified. Note that if you use this form and leave out `Prelude`, an implicit `Prelude` import will be added automatically.

The *:module* (page 51) command provides a way to do two things that cannot be done with ordinary `import` declarations:

- *:module* (page 51) supports the * modifier on modules, which opens the full top-level scope of a module, rather than just its exports.
- Imports can be *removed* from the context, using the syntax `:module -M`. The `import` syntax is cumulative (as in a Haskell module), so this is the only way to subtract from the scope.

3.4.5.4 Qualified names

-fimplicit-import-qualified

Default on

To make life slightly easier, the GHCi prompt also behaves as if there is an implicit `import qualified` declaration for every module in every package, and every module currently loaded into GHCi. This behaviour can be disabled with the `-fno-implicit-import-qualified` flag.

3.4.5.5 `:module` and `:load`

It might seem that *`:module`* (page 51)/`import` and *`:load`* (page 50)/*`:add`* (page 44)/*`:reload`* (page 51) do similar things: you can use both to bring a module into scope. However, there is a very important difference. GHCi is concerned with two sets of modules:

- The set of modules that are currently *loaded*. This set is modified by *`:load`* (page 50), *`:reload`* (page 51), *`:add`* (page 44) and *`:unadd`* (page 55), and can be shown with *`:show modules`* (page 54).
- The set of modules that are currently *in scope* at the prompt. This set is modified by `import` and *`:module`* (page 51), and it is also modified automatically after *`:load`* (page 50), *`:reload`* (page 51), *`:add`* (page 44) and *`:unadd`* (page 55), as described above. The set of modules in scope can be shown with *`:show imports`* (page 53).

You can add a module to the scope (via *`:module`* (page 51) or `import`) only if either (a) it is loaded, or (b) it is a module from a package that GHCi knows about. Using *`:module`* (page 51) or `import` to try bring into scope a non-loaded module may result in the message `module M is not loaded`.

3.4.5.6 Shadowing and the `Ghci1` module name

Bindings on the prompt can shadow earlier bindings:

```
ghci> let foo = True
ghci> let foo = False
ghci> :show bindings
foo :: Bool = False
```

But the shadowed thing still exists, and may show up again later, for example in a type signature:

```
ghci> data T = A | B deriving Eq
ghci> let a = A
ghci> data T = ANewType
ghci> :t a
a :: Ghci1.T
```

Now the type of `a` is printed using the fully qualified name of `T`, using the module name `Ghci1` (and `Ghci2` for the next set of bindings, and so on). You can use these qualified names as well:

```
ghci> a == Ghci1.A
True
ghci> let a = False -- shadowing a
ghci> Ghci2.a == Ghci1.A
True
```

The command `:show bindings` only shows bindings that are not shadowed. Bindings that define multiple names, such as a type constructor and its data constructors, are shown if *any* defined name is still available without the need for qualification.

3.4.6 The it variable

Whenever an expression (or a non-binding statement, to be precise) is typed at the prompt, GHCi implicitly binds its value to the variable `it`. For example:

```
ghci> 1+2
3
ghci> it * 2
6
```

What actually happens is that GHCi typechecks the expression, and if it doesn't have an `IO` type, then it transforms it as follows: an expression `e` turns into

```
let it = e;
print it
```

which is then run as an IO-action.

Hence, the original expression must have a type which is an instance of the `Show` class, or GHCi will complain:

```
ghci> id

<interactive>:1:0:
    No instance for (Show (a -> a))
      arising from use of `print' at <interactive>:1:0-1
    Possible fix: add an instance declaration for (Show (a -> a))
    In the expression: print it
    In a 'do' expression: print it
```

The error message contains some clues as to the transformation happening internally.

If the expression was instead of type `IO a` for some `a`, then `it` will be bound to the result of the `IO` computation, which is of type `a`. eg.:

```
ghci> Data.Time.getZonedTime
2017-04-10 12:34:56.93213581 UTC
ghci> print it
2017-04-10 12:34:56.93213581 UTC
```

The corresponding translation for an IO-typed `e` is

```
it <- e
```

Note that `it` is shadowed by the new value each time you evaluate a new expression, and the old value of `it` is lost.

In order to stop the value `it` being bound on each command, the flag *-fno-it* (page 28) can be set. The `it` variable can be the source of space leaks due to how shadowed declarations are handled by GHCi (see *Type, class and other declarations* (page 23)).

-fno-it

When this flag is set, the variable `it` will no longer be set to the result of the previously evaluated expression.

3.4.7 Type defaulting in GHCi

ExtendedDefaultRules

Since 6.8.1

Allow defaulting to take place for more than just numeric classes.

Consider this GHCi session:

```
ghci> reverse []
```

What should GHCi do? Strictly speaking, the program is ambiguous. `show (reverse [])` (which is what GHCi computes here) has type `Show a => String` and how that displays depends on the type `a`. For example:

```
ghci> reverse ([] :: String)
""
ghci> reverse ([] :: [Int])
[]
```

However, it is tiresome for the user to have to specify the type, so GHCi extends Haskell's type-defaulting rules (Section 4.3.4 of the Haskell 2010 Report) as follows. The standard rules take each group of constraints `(C1 a, C2 a, ..., Cn a)` for each type variable `a`, and defaults the type variable if

1. The type variable `a` appears in no other constraints
2. All the classes `Ci` are standard.
3. At least one of the classes `Ci` is numeric.

At the GHCi prompt, or with GHC if the *ExtendedDefaultRules* (page 29) flag is given, the types are instead resolved with the following method:

Find all the unsolved constraints. Then:

- Find those that are of form `(C a)` where `a` is a type variable, and partition those constraints into groups that share a common type variable `a`.
- Keep only the groups in which at least one of the classes is an **interactive class** (defined below).
- Now, for each remaining group G, try each type `ty` from the default-type list in turn; if setting `a = ty` would allow the constraints in G to be completely solved. If so, default `a` to `ty`.
- The unit type `()` and the list type `[]` are added to the start of the standard list of types which are tried when doing type defaulting.

Note that any multi-parameter constraints `(D a b)` or `(D [a] Int)` do not participate in the process (either to help or to hinder); but they must of course be soluble once the defaulting process is complete.

The last point means that, for example, this program:

```
main :: IO ()
main = print def

instance Num ()
```

(continues on next page)

(continued from previous page)

```
def :: (Num a, Enum a) => a
def = toEnum 0
```

prints `()` rather than `0` as the type is defaulted to `()` rather than `Integer`.

The motivation for the change is that it means `IO a` actions default to `IO ()`, which in turn means that ghci won't try to print a result when running them. This is particularly important for `printf`, which has an instance that returns `IO a`. However, it is only able to return `undefined` (the reason for the instance having this type is so that printf doesn't require extensions to the class system), so if the type defaults to `Integer` then ghci gives an error when running a printf.

See also *I/O actions at the prompt* (page 20) for how the monad of a computational expression defaults to `IO` if possible.

3.4.7.1 Interactive classes

The interactive classes (only relevant when *`ExtendedDefaultRules`* (page 29) is in effect) are: any numeric class, `Show`, `Eq`, `Ord`, `Foldable` or `Traversable`.

As long as a type variable is constrained by one of these classes, defaulting will occur, as outlined above.

3.4.7.2 Extended rules around `default` declarations

Since the rules for defaulting are relaxed under *`ExtendedDefaultRules`* (page 29), the rules for `default` declarations are also relaxed. According to Section 4.3.4 of the Haskell 2010 Report, a `default` declaration looks like `default (t1, ..., tn)` where, for each `ti`, `Num ti` must hold. This is relaxed to say that for each `ti`, there must exist an interactive class `C` such that `C ti` holds. This means that type *constructors* can be allowed in these lists. For example, the following works if you wish your `Foldable` constraints to default to `Maybe` but your `Num` constraints to still default to `Integer` or `Double`:

```
default (Maybe, Integer, Double)
```

3.4.8 Using a custom interactive printing function

Since GHC 7.6.1, GHCi prints the result of expressions typed at the prompt using the function `System.IO.print`. Its type signature is `Show a => a -> IO ()`, and it works by converting the value to `String` using `show`.

This is not ideal in certain cases, like when the output is long, or contains strings with non-ascii characters.

The *`-interactive-print ⟨name⟩`* (page 30) flag allows to specify any function of type `C a => a -> IO ()`, for some constraint `C`, as the function for printing evaluated expressions. The function can reside in any loaded module or any registered package, but only when it resides in a registered package will it survive a *`:cd`* (page 45), *`:add`* (page 44), *`:unadd`* (page 55), *`:load`* (page 50), *`:reload`* (page 51) or, *`:set`* (page 52).

`-interactive-print` ⟨name⟩
: Set the function used by GHCi to print evaluation results. Given name must be of type `C a => a -> IO ()`.

As an example, suppose we have following special printing module:

```
module SpecPrinter where
import System.IO

sprint a = putStrLn $ show a ++ "!"
```

The `sprint` function adds an exclamation mark at the end of any printed value. Running GHCi with the command:

```
ghci -interactive-print=SpecPrinter.sprint SpecPrinter
```

will start an interactive session where values with be printed using `sprint`:

```
*SpecPrinter> [1,2,3]
[1,2,3]!
*SpecPrinter> 42
42!
```

A custom pretty printing function can be used, for example, to format tree-like and nested structures in a more readable way.

The *-interactive-print ⟨name⟩* (page 30) flag can also be used when running GHC in `-e mode`:

```
% ghc -e "[1,2,3]" -interactive-print=SpecPrinter.sprint SpecPrinter
[1,2,3]!
```

3.4.9 Stack Traces in GHCi

[This is an experimental feature enabled by the new `-fexternal-interpreter` flag that was introduced in GHC 8.0.1. It is currently not supported on Windows.]

GHCi can use the profiling system to collect stack trace information when running interpreted code. To gain access to stack traces, start GHCi like this:

```
ghci -fexternal-interpreter -prof
```

This runs the interpreted code in a separate process (see *Running the interpreter in a separate process* (page 60)) and runs it in profiling mode to collect call stack information. Note that because we're running the interpreted code in profiling mode, all packages that you use must be compiled for profiling. The `-prof` flag to GHCi only works in conjunction with `-fexternal-interpreter`.

There are three ways to get access to the current call stack.

- `error` and `undefined` automatically attach the current stack to the error message. This often complements the `HasCallStack` stack (see *HasCallStack* (page 602)), so both call stacks are shown.
- `Debug.Trace.traceStack` is a version of `Debug.Trace.trace` that also prints the current call stack.
- Functions in the module `GHC.Stack` can be used to get the current stack and render it.

You don't need to use `-fprof-auto` for interpreted modules, annotations are automatically added at a granularity fine enough to distinguish individual call sites. However, you won't see any call stack information for compiled code unless it was compiled with `-fprof-auto` or has explicit `SCC` annotations (see *Inserting cost centres by hand* (page 654)).

3.5 The GHCi Debugger

GHCi contains a simple imperative-style debugger in which you can stop a running computation in order to examine the values of variables. The debugger is integrated into GHCi, and is turned on by default: no flags are required to enable the debugging facilities. There is one major restriction: breakpoints and single-stepping are only available in interpreted modules; compiled code is invisible to the debugger[5].

The debugger provides the following:

- The ability to set a breakpoint on a function definition or expression in the program. When the function is called, or the expression evaluated, GHCi suspends execution and returns to the prompt, where you can inspect the values of local variables before continuing with the execution.
- Execution can be single-stepped: the evaluator will suspend execution approximately after every reduction, allowing local variables to be inspected. This is equivalent to setting a breakpoint at every point in the program.
- Execution can take place in tracing mode, in which the evaluator remembers each evaluation step as it happens, but doesn't suspend execution until an actual breakpoint is reached. When this happens, the history of evaluation steps can be inspected.
- Exceptions (e.g. pattern matching failure and `error`) can be treated as breakpoints, to help locate the source of an exception in the program.

There is currently no support for obtaining a "stack trace", but the tracing and history features provide a useful second-best, which will often be enough to establish the context of an error. For instance, it is possible to break automatically when an exception is thrown, even if it is thrown from within compiled code (see *Debugging exceptions* (page 40)).

`-fbreak-points`

Default enabled for GHCi

This flag's purpose is to allow disabling breakpoint insertion with the reverse form.

3.5.1 Breakpoints and inspecting variables

Let's use quicksort as a running example. Here's the code:

```
qsort [] = []
qsort (a:as) = qsort left ++ [a] ++ qsort right
  where (left,right) = (filter (<=a) as, filter (>a) as)

main = print (qsort [8, 4, 0, 3, 1, 23, 11, 18])
```

First, load the module into GHCi:

[5] Note that packages only contain compiled code, so debugging a package requires finding its source and loading that directly.

```
ghci> :l qsort.hs
[1 of 1] Compiling Main             ( qsort.hs, interpreted )
Ok, modules loaded: Main.
*ghci>
```

Now, let's set a breakpoint on the right-hand-side of the second equation of qsort:

```
*ghci> :break 2
Breakpoint 0 activated at qsort.hs:2:15-46
*ghci>
```

The command `:break 2` sets a breakpoint on line 2 of the most recently-loaded module, in this case `qsort.hs`. Specifically, it picks the leftmost complete subexpression on that line on which to set the breakpoint, which in this case is the expression (`qsort left ++ [a] ++ qsort right`).

Now, we run the program:

```
*ghci> main
Stopped at qsort.hs:2:15-46
_result :: [a]
a :: a
left :: [a]
right :: [a]
[qsort.hs:2:15-46] *ghci>
```

Execution has stopped at the breakpoint. The prompt has changed to indicate that we are currently stopped at a breakpoint, and the location: `[qsort.hs:2:15-46]`. To further clarify the location, we can use the *:list* (page 50) command:

```
[qsort.hs:2:15-46] *ghci> :list
1  qsort [] = []
2  qsort (a:as) = qsort left ++ [a] ++ qsort right
3    where (left,right) = (filter (<=a) as, filter (>a) as)
```

The *:list* (page 50) command lists the source code around the current breakpoint. If your output device supports it, then GHCi will highlight the active subexpression in bold.

GHCi has provided bindings for the free variables[6] of the expression on which the breakpoint was placed (`a`, `left`, `right`), and additionally a binding for the result of the expression (`_result`). These variables are just like other variables that you might define in GHCi; you can use them in expressions that you type at the prompt, you can ask for their types with *:type* (page 54), and so on. There is one important difference though: these variables may only have partial types. For example, if we try to display the value of `left`:

```
[qsort.hs:2:15-46] *ghci> left

<interactive>:1:0:
    Ambiguous type variable `a' in the constraint:
      `Show a' arising from a use of `print' at <interactive>:1:0-3
    Cannot resolve unknown runtime types: a
    Use :print or :force to determine these types
```

This is because `qsort` is a polymorphic function, and because GHCi does not carry type information at runtime, it cannot determine the runtime types of free variables that involve type

[6] We originally provided bindings for all variables in scope, rather than just the free variables of the expression, but found that this affected performance considerably, hence the current restriction to just the free variables.

variables. Hence, when you ask to display `left` at the prompt, GHCi can't figure out which instance of `Show` to use, so it emits the type error above.

Fortunately, the debugger includes a generic printing command, *:print* (page 51), which can inspect the actual runtime value of a variable and attempt to reconstruct its type. If we try it on `left`:

```
[qsort.hs:2:15-46] *ghci> :set -fprint-evld-with-show
[qsort.hs:2:15-46] *ghci> :print left
left = (_t1::[a])
```

This isn't particularly enlightening. What happened is that `left` is bound to an unevaluated computation (a suspension, or thunk), and *:print* (page 51) does not force any evaluation. The idea is that *:print* (page 51) can be used to inspect values at a breakpoint without any unfortunate side effects. It won't force any evaluation, which could cause the program to give a different answer than it would normally, and hence it won't cause any exceptions to be raised, infinite loops, or further breakpoints to be triggered (see *Nested breakpoints* (page 37)). Rather than forcing thunks, *:print* (page 51) binds each thunk to a fresh variable beginning with an underscore, in this case `_t1`.

-fprint-evld-with-show

The flag *-fprint-evld-with-show* (page 34) instructs *:print* (page 51) to reuse available `Show` instances when possible. This happens only when the contents of the variable being inspected are completely evaluated.

If we aren't concerned about preserving the evaluatedness of a variable, we can use *:force* (page 48) instead of *:print* (page 51). The *:force* (page 48) command behaves exactly like *:print* (page 51), except that it forces the evaluation of any thunks it encounters:

```
[qsort.hs:2:15-46] *ghci> :force left
left = [4,0,3,1]
```

Now, since *:force* (page 48) has inspected the runtime value of `left`, it has reconstructed its type. We can see the results of this type reconstruction:

```
[qsort.hs:2:15-46] *ghci> :show bindings
_result :: [Integer]
a :: Integer
left :: [Integer]
right :: [Integer]
_t1 :: [Integer]
```

Not only do we now know the type of `left`, but all the other partial types have also been resolved. So we can ask for the value of `a`, for example:

```
[qsort.hs:2:15-46] *ghci> a
8
```

You might find it useful to use Haskell's `seq` function to evaluate individual thunks rather than evaluating the whole expression with *:force* (page 48). For example:

```
[qsort.hs:2:15-46] *ghci> :print right
right = (_t1::[Integer])
[qsort.hs:2:15-46] *ghci> seq _t1 ()
()
[qsort.hs:2:15-46] *ghci> :print right
right = 23 : (_t2::[Integer])
```

We evaluated only the `_t1` thunk, revealing the head of the list, and the tail is another thunk now bound to `_t2`. The `seq` function is a little inconvenient to use here, so you might want to use *:def* (page 46) to make a nicer interface (left as an exercise for the reader!).

Finally, we can continue the current execution:

```
[qsort.hs:2:15-46] *ghci> :continue
Stopped at qsort.hs:2:15-46
_result :: [a]
a :: a
left :: [a]
right :: [a]
[qsort.hs:2:15-46] *ghci>
```

The execution continued at the point it previously stopped, and has now stopped at the breakpoint for a second time.

3.5.1.1 Setting breakpoints

Breakpoints can be set in various ways. Perhaps the easiest way to set a breakpoint is to name a top-level function:

```
:break identifier
```

Where ⟨identifier⟩ names any top-level function in an interpreted module currently loaded into GHCi (qualified names may be used). The breakpoint will be set on the body of the function, when it is fully applied. If the function has several patterns, then a breakpoint will be set on each of them.

By using qualified names, one can set breakpoints on all functions (top-level and nested) in every loaded and interpreted module:

```
:break [ModQual.]topLevelIdent[.nestedIdent]...[.nestedIdent]
```

⟨ModQual⟩ is optional and is either the effective name of a module or the local alias of a qualified import statement.

⟨topLevelIdent⟩ is the name of a top level function in the module referenced by ⟨ModQual⟩.

⟨nestedIdent⟩ is optional and the name of a function nested in a let or where clause inside the previously mentioned function ⟨nestedIdent⟩ or ⟨topLevelIdent⟩.

If ⟨ModQual⟩ is a module name, then ⟨topLevelIdent⟩ can be any top level identifier in this module. If ⟨ModQual⟩ is missing or a local alias of a qualified import, then ⟨topLevelIdent⟩ must be in scope.

Breakpoints can be set on arbitrarily deeply nested functions, but the whole chain of nested function names must be specified.

Consider the function `foo` in a module `Main`:

```
foo s = 'a' : add s
    where add = (++"z")
```

The breakpoint on the function `add` can be set with one of the following commands:

```
:break Main.foo.add
:break foo.add
```

Breakpoints can also be set by line (and optionally column) number:

```
:break line
:break line column
:break module line
:break module line column
```

When a breakpoint is set on a particular line, GHCi sets the breakpoint on the leftmost subexpression that begins and ends on that line. If two complete subexpressions start at the same column, the longest one is picked. If there is no complete subexpression on the line, then the leftmost expression starting on the line is picked, and failing that the rightmost expression that partially or completely covers the line.

When a breakpoint is set on a particular line and column, GHCi picks the smallest subexpression that encloses that location on which to set the breakpoint. Note: GHC considers the TAB character to have a width of 1, wherever it occurs; in other words it counts characters, rather than columns. This matches what some editors do, and doesn't match others. The best advice is to avoid tab characters in your source code altogether (see *-Wtabs* (page 101) in *Warnings and sanity-checking* (page 84)).

If the module is omitted, then the most recently-loaded module is used.

Not all subexpressions are potential breakpoint locations. Single variables are typically not considered to be breakpoint locations (unless the variable is the right-hand-side of a function definition, lambda, or case alternative). The rule of thumb is that all redexes are breakpoint locations, together with the bodies of functions, lambdas, case alternatives and binding statements. There is normally no breakpoint on a let expression, but there will always be a breakpoint on its body, because we are usually interested in inspecting the values of the variables bound by the let.

3.5.1.2 Managing breakpoints

The list of breakpoints currently defined can be displayed using *:show breaks* (page 53):

```
*ghci> :show breaks
[0] Main qsort.hs:1:11-12 enabled
[1] Main qsort.hs:2:15-46 enabled
```

To disable one or several defined breakpoint, use the *:disable* (page 47) command with one or several blank separated numbers given in the output from *:show breaks* (page 53):. To disable all breakpoints at once, use `:disable *`.

```
*ghci> :disable 0
*ghci> :show breaks
[0] Main qsort.hs:1:11-12 disabled
[1] Main qsort.hs:2:15-46 enabled
```

Disabled breakpoints can be (re-)enabled with the *:enable* (page 47) command. The parameters of the *:disable* (page 47) and *:enable* (page 47) commands are identical.

To delete a breakpoint, use the *:delete* (page 47) command with the number given in the output from *:show breaks* (page 53):

```
*ghci> :delete 0
*ghci> :show breaks
[1] Main qsort.hs:2:15-46 disabled
```

To delete all breakpoints at once, use `:delete *`.

3.5.2 Single-stepping

Single-stepping is a great way to visualise the execution of your program, and it is also a useful tool for identifying the source of a bug. GHCi offers two variants of stepping. Use *:step* (page 54) to enable all the breakpoints in the program, and execute until the next breakpoint is reached. Use *:steplocal* (page 54) to limit the set of enabled breakpoints to those in the current top level function. Similarly, use *:stepmodule* (page 54) to single step only on breakpoints contained in the current module. For example:

```
*ghci> :step main
Stopped at qsort.hs:5:7-47
_result :: IO ()
```

The command *:step expr* (page 54) begins the evaluation of ⟨expr⟩ in single-stepping mode. If ⟨expr⟩ is omitted, then it single-steps from the current breakpoint. *:steplocal* (page 54) and *:stepmodule* (page 54) commands work similarly.

The *:list* (page 50) command is particularly useful when single-stepping, to see where you currently are:

```
[qsort.hs:5:7-47] *ghci> :list
4
5  main = print (qsort [8, 4, 0, 3, 1, 23, 11, 18])
6
[qsort.hs:5:7-47] *ghci>
```

In fact, GHCi provides a way to run a command when a breakpoint is hit, so we can make it automatically do *:list* (page 50):

```
[qsort.hs:5:7-47] *ghci> :set stop :list
[qsort.hs:5:7-47] *ghci> :step
Stopped at qsort.hs:5:14-46
_result :: [Integer]
4
5  main = print (qsort [8, 4, 0, 3, 1, 23, 11, 18])
6
[qsort.hs:5:14-46] *ghci>
```

3.5.3 Nested breakpoints

When GHCi is stopped at a breakpoint, and an expression entered at the prompt triggers a second breakpoint, the new breakpoint becomes the "current" one, and the old one is saved on a stack. An arbitrary number of breakpoint contexts can be built up in this way. For example:

```
[qsort.hs:2:15-46] *ghci> :st qsort [1,3]
Stopped at qsort.hs:(1,0)-(3,55)
_result :: [a]
... [qsort.hs:(1,0)-(3,55)] *ghci>
```

While stopped at the breakpoint on line 2 that we set earlier, we started a new evaluation with `:step qsort [1,3]`. This new evaluation stopped after one step (at the definition of `qsort`).

The prompt has changed, now prefixed with `...`, to indicate that there are saved breakpoints beyond the current one. To see the stack of contexts, use *:show context* (page 53):

```
... [qsort.hs:(1,0)-(3,55)] *ghci> :show context
--> main
  Stopped at qsort.hs:2:15-46
--> qsort [1,3]
  Stopped at qsort.hs:(1,0)-(3,55)
... [qsort.hs:(1,0)-(3,55)] *ghci>
```

To abandon the current evaluation, use *:abandon* (page 44):

```
... [qsort.hs:(1,0)-(3,55)] *ghci> :abandon
[qsort.hs:2:15-46] *ghci> :abandon
*ghci>
```

3.5.4 The _result variable

When stopped at a breakpoint or single-step, GHCi binds the variable `_result` to the value of the currently active expression. The value of `_result` is presumably not available yet, because we stopped its evaluation, but it can be forced: if the type is known and showable, then just entering `_result` at the prompt will show it. However, there's one caveat to doing this: evaluating `_result` will be likely to trigger further breakpoints, starting with the breakpoint we are currently stopped at (if we stopped at a real breakpoint, rather than due to *:step* (page 54)). So it will probably be necessary to issue a *:continue* (page 46) immediately when evaluating `_result`. Alternatively, you can use *:force* (page 48) which ignores breakpoints.

3.5.5 Tracing and history

A question that we often want to ask when debugging a program is "how did I get here?". Traditional imperative debuggers usually provide some kind of stack-tracing feature that lets you see the stack of active function calls (sometimes called the "lexical call stack"), describing a path through the code to the current location. Unfortunately this is hard to provide in Haskell, because execution proceeds on a demand-driven basis, rather than a depth-first basis as in strict languages. The "stack" in GHC's execution engine bears little resemblance to the lexical call stack. Ideally GHCi would maintain a separate lexical call stack in addition to the dynamic call stack, and in fact this is exactly what our profiling system does (*Profiling* (page 651)), and what some other Haskell debuggers do. For the time being, however, GHCi doesn't maintain a lexical call stack (there are some technical challenges to be overcome). Instead, we provide a way to backtrack from a breakpoint to previous evaluation steps: essentially this is like single-stepping backwards, and should in many cases provide enough information to answer the "how did I get here?" question.

To use tracing, evaluate an expression with the *:trace* (page 54) command. For example, if we set a breakpoint on the base case of `qsort`:

```
*ghci> :list qsort
1  qsort [] = []
2  qsort (a:as) = qsort left ++ [a] ++ qsort right
3    where (left,right) = (filter (<=a) as, filter (>a) as)
4
*ghci> :b 1
Breakpoint 1 activated at qsort.hs:1:11-12
*ghci>
```

and then run a small `qsort` with tracing:

```
*ghci> :trace qsort [3,2,1]
Stopped at qsort.hs:1:11-12
_result :: [a]
[qsort.hs:1:11-12] *ghci>
```

We can now inspect the history of evaluation steps:

```
[qsort.hs:1:11-12] *ghci> :hist
-1  : qsort.hs:3:24-38
-2  : qsort.hs:3:23-55
-3  : qsort.hs:(1,0)-(3,55)
-4  : qsort.hs:2:15-24
-5  : qsort.hs:2:15-46
-6  : qsort.hs:3:24-38
-7  : qsort.hs:3:23-55
-8  : qsort.hs:(1,0)-(3,55)
-9  : qsort.hs:2:15-24
-10 : qsort.hs:2:15-46
-11 : qsort.hs:3:24-38
-12 : qsort.hs:3:23-55
-13 : qsort.hs:(1,0)-(3,55)
-14 : qsort.hs:2:15-24
-15 : qsort.hs:2:15-46
-16 : qsort.hs:(1,0)-(3,55)
<end of history>
```

To examine one of the steps in the history, use *:back* (page 44):

```
[qsort.hs:1:11-12] *ghci> :back
Logged breakpoint at qsort.hs:3:24-38
_result :: [a]
as :: [a]
a :: a
[-1: qsort.hs:3:24-38] *ghci>
```

Note that the local variables at each step in the history have been preserved, and can be examined as usual. Also note that the prompt has changed to indicate that we're currently examining the first step in the history: `-1`. The command *:forward* (page 48) can be used to traverse forward in the history.

The *:trace* (page 54) command can be used with or without an expression. When used without an expression, tracing begins from the current breakpoint, just like *:step* (page 54).

The history is only available when using *:trace* (page 54); the reason for this is we found that logging each breakpoint in the history cuts performance by a factor of 2 or more.

-fghci-hist-size=⟨n⟩

Default 50

Modify the depth of the evaluation history tracked by GHCi.

3.5.6 Debugging exceptions

Another common question that comes up when debugging is "where did this exception come from?". Exceptions such as those raised by `error` or `head []` have no context information attached to them. Finding which particular call to `head` in your program resulted in the error can be a painstaking process, usually involving `Debug.Trace.trace`, or compiling with profiling and using `Debug.Trace.traceStack` or `+RTS -xc` (see *-xc* (page 198)).

The GHCi debugger offers a way to hopefully shed some light on these errors quickly and without modifying or recompiling the source code. One way would be to set a breakpoint on the location in the source code that throws the exception, and then use *:trace* (page 54) and *:history* (page 48) to establish the context. However, `head` is in a library and we can't set a breakpoint on it directly. For this reason, GHCi provides the flags *-fbreak-on-exception* (page 40) which causes the evaluator to stop when an exception is thrown, and *-fbreak-on-error* (page 40), which works similarly but stops only on uncaught exceptions. When stopping at an exception, GHCi will act just as it does when a breakpoint is hit, with the deviation that it will not show you any source code location. Due to this, these commands are only really useful in conjunction with *:trace* (page 54), in order to log the steps leading up to the exception. For example:

```
*ghci> :set -fbreak-on-exception
*ghci> :trace qsort ("abc" ++ undefined)
“Stopped at <exception thrown>
_exception :: e
[<exception thrown>] *ghci> :hist
-1  : qsort.hs:3:24-38
-2  : qsort.hs:3:23-55
-3  : qsort.hs:(1,0)-(3,55)
-4  : qsort.hs:2:15-24
-5  : qsort.hs:2:15-46
-6  : qsort.hs:(1,0)-(3,55)
<end of history>
[<exception thrown>] *ghci> :back
Logged breakpoint at qsort.hs:3:24-38
_result :: [a]
as :: [a]
a :: a
[-1: qsort.hs:3:24-38] *ghci> :force as
*** Exception: Prelude.undefined
[-1: qsort.hs:3:24-38] *ghci> :print as
as = 'b' : 'c' : (_t1::[Char])
```

The exception itself is bound to a new variable, `_exception`.

Breaking on exceptions is particularly useful for finding out what your program was doing when it was in an infinite loop. Just hit Control-C, and examine the history to find out what was going on.

-fbreak-on-exception
: Causes GHCi to halt evaluation and return to the interactive prompt in the event of an exception. *-fbreak-on-exception* (page 40) breaks on all exceptions.

-fbreak-on-error
: Causes GHCi to halt evaluation and return to the interactive prompt in the event of an exception. *-fbreak-on-error* (page 40) breaks on only those exceptions which would otherwise be uncaught.

3.5.7 Example: inspecting functions

It is possible to use the debugger to examine function values. When we are at a breakpoint and a function is in scope, the debugger cannot show you the source code for it; however, it is possible to get some information by applying it to some arguments and observing the result.

The process is slightly complicated when the binding is polymorphic. We show the process by means of an example. To keep things simple, we will use the well known `map` function:

```
import Prelude hiding (map)

map :: (a->b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

We set a breakpoint on `map`, and call it.

```
*ghci> :break 5
Breakpoint 0 activated at  map.hs:5:15-28
*ghci> map Just [1..5]
Stopped at map.hs:(4,0)-(5,12)
_result :: [b]
x :: a
f :: a -> b
xs :: [a]
```

GHCi tells us that, among other bindings, `f` is in scope. However, its type is not fully known yet, and thus it is not possible to apply it to any arguments. Nevertheless, observe that the type of its first argument is the same as the type of `x`, and its result type is shared with `_result`.

As we demonstrated earlier (*Breakpoints and inspecting variables* (page 32)), the debugger has some intelligence built-in to update the type of `f` whenever the types of `x` or `_result` are discovered. So what we do in this scenario is force `x` a bit, in order to recover both its type and the argument part of `f`.

```
*ghci> seq x ()
*ghci> :print x
x = 1
```

We can check now that as expected, the type of `x` has been reconstructed, and with it the type of `f` has been too:

```
*ghci> :t x
x :: Integer
*ghci> :t f
f :: Integer -> b
```

From here, we can apply f to any argument of type Integer and observe the results.

```
*ghci> let b = f 10
*ghci> :t b
b :: b
*ghci> b
<interactive>:1:0:
    Ambiguous type variable `b' in the constraint:
      `Show b' arising from a use of `print' at <interactive>:1:0
```

(continues on next page)

(continued from previous page)

```
*ghci> :p b
b = (_t2::a)
*ghci> seq b ()
()
*ghci> :t b
b :: a
*ghci> :p b
b = Just 10
*ghci> :t b
b :: Maybe Integer
*ghci> :t f
f :: Integer -> Maybe Integer
*ghci> f 20
Just 20
*ghci> map f [1..5]
[Just 1, Just 2, Just 3, Just 4, Just 5]
```

In the first application of `f`, we had to do some more type reconstruction in order to recover the result type of `f`. But after that, we are free to use `f` normally.

3.5.8 Limitations

- When stopped at a breakpoint, if you try to evaluate a variable that is already under evaluation, the second evaluation will hang. The reason is that GHC knows the variable is under evaluation, so the new evaluation just waits for the result before continuing, but of course this isn’t going to happen because the first evaluation is stopped at a breakpoint. Control-C can interrupt the hung evaluation and return to the prompt.

  The most common way this can happen is when you’re evaluating a CAF (e.g. main), stop at a breakpoint, and ask for the value of the CAF at the prompt again.

- Implicit parameters (see *Implicit parameters* (page 517)) are only available at the scope of a breakpoint if there is an explicit type signature.

3.6 Invoking GHCi

GHCi is invoked with the command `ghci` or `ghc --interactive`. One or more modules or filenames can also be specified on the command line; this instructs GHCi to load the specified modules or filenames (and all the modules they depend on), just as if you had said `:load modules` at the GHCi prompt (see *GHCi commands* (page 44)). For example, to start GHCi and load the program whose topmost module is in the file `Main.hs`, we could say:

```
$ ghci Main.hs
```

Most of the command-line options accepted by GHC (see *Using GHC* (page 65)) also make sense in interactive mode. The ones that don’t make sense are mostly obvious.

-flocal-ghci-history

By default, GHCi keeps global history in `$XDG_DATA_HOME/ghc/ghci_history` or `%APPDATA%/<app>/ghci_history`, but you can use current directory, e.g.:

```
$ ghci -flocal-ghci-history
```

It will create `.ghci-history` in current folder where GHCi is launched.

-fghci-leak-check
(Debugging only) When loading new modules with `:load`, check that any previously loaded modules have been correctly garbage collected. Emits messages if a leak is detected.

3.6.1 Packages

Most packages (see *Using Packages* (page 219)) are available without needing to specify any extra flags at all: they will be automatically loaded the first time they are needed.

For hidden packages, however, you need to request the package be loaded by using the *-package ⟨pkg⟩* (page 221) flag:

```
$ ghci -package readline
GHCi, version 8.y.z: https://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
Loading package readline-1.0 ... linking ... done.
ghci>
```

The following command works to load new packages into a running GHCi:

```
ghci> :set -package name
```

But note that doing this will cause all currently loaded modules to be unloaded, and you'll be dumped back into the `Prelude`.

3.6.2 Extra libraries

Extra libraries may be specified on the command line using the normal `-llib` option. (The term *library* here refers to libraries of foreign object code; for using libraries of Haskell source code, see *Modules vs. filenames* (page 17).) For example, to load the "m" library:

```
$ ghci -lm
```

On systems with `.so`-style shared libraries, the actual library loaded will the `liblib.so`. GHCi searches the following places for libraries, in this order:

- Paths specified using the *-L ⟨dir⟩* (page 246) command-line option,
- The standard library search path for your system loader, which on some systems may be overridden by setting the `LD_LIBRARY_PATH` environment variable.
- The linker standard library search can also be overridden on some systems using the `LIBRARY_PATH` environment variable. Because of some implementation detail on Windows, setting `LIBRARY_PATH` will also extend the system loader path for any library it finds. So often setting `LIBRARY_PATH` is enough.

On systems with `.dll`-style shared libraries, the actual library loaded will be `lib.dll`, `liblib.dll`. GHCi also has full support for import libraries, either Microsoft style `.lib`, or GNU GCC style `.a` and `.dll.a` libraries. If you have an import library it is advisable to always specify the import library instead of the `.dll`. e.g. use `-lgcc` instead of ``-llibgcc_s_seh-1``. Again, GHCi will signal an error if it can't find the library.

GHCi can also load plain object files (`.o` or `.obj` depending on your platform) or static archives (`.a`) from the command-line. Just add the name the object file or library to the command line. On Windows GHCi also supports the `big-obj` format.

Ordering of `-l` options matters: a library should be mentioned *before* the libraries it depends on (see *Options affecting linking* (page 245)).

3.7 GHCi commands

GHCi commands all begin with ":" and consist of a single command name followed by zero or more parameters. The command name may be abbreviated, with ambiguities being resolved in favour of the more commonly used commands.

:abandon
: Abandons the current evaluation (only available when stopped at a breakpoint).

:add `[*]⟨module⟩`
: Add ⟨module⟩(s) to the current target set, and perform a reload. Normally pre-compiled code for the module will be loaded if available, or otherwise the module will be compiled to byte-code. Using the * prefix forces the module to be loaded as byte-code.

  ⟨module⟩ may be a file path. A "~" symbol at the beginning of ⟨module⟩ will be replaced by the contents of the environment variable `HOME`.

:all-types
: List all types collected for expressions and (local) bindings currently loaded (while *:set +c* (page 56) was active) with their respective source-code span, e.g.

```
GhciTypes> :all-types
GhciTypes.hs:(38,13)-(38,24): Maybe Id
GhciTypes.hs:(45,10)-(45,29): Outputable SpanInfo
GhciTypes.hs:(45,10)-(45,29): (Rational -> SpanInfo -> SDoc) -> Outputable␣
↪SpanInfo
```

:back `⟨n⟩`
: Travel back ⟨n⟩ steps in the history. ⟨n⟩ is one if omitted. See *Tracing and history* (page 38) for more about GHCi's debugging facilities. See also: *:trace* (page 54), *:history* (page 48), *:forward* (page 48).

:break `[⟨identifier⟩ | [⟨module⟩] ⟨line⟩ [⟨column⟩]]`
: Set a breakpoint on the specified function or line and column. See *Setting breakpoints* (page 35).

:browse`[!] [[*] ⟨module⟩]`
: Displays the identifiers exported by the module ⟨module⟩, which must be either loaded into GHCi or be a member of a package. If ⟨module⟩ is omitted, the most recently-loaded module is used.

  Like all other GHCi commands, the output is always displayed in the current GHCi scope (*What's really in scope at the prompt?* (page 24)).

  There are two variants of the browse command:

  - If the * symbol is placed before the module name, then *all* the identifiers in scope in ⟨module⟩ (rather that just its exports) are shown.

    The *-form is only available for modules which are interpreted; for compiled modules (including modules from packages) only the non-* form of *:browse* (page 44) is available.

- Data constructors and class methods are usually displayed in the context of their data type or class declaration. However, if the ! symbol is appended to the command, thus `:browse!`, they are listed individually. The `!`-form also annotates the listing with comments giving possible imports for each group of entries. Here is an example:

```
ghci> :browse! Data.Maybe
-- not currently imported
Data.Maybe.catMaybes :: [Maybe a] -> [a]
Data.Maybe.fromJust :: Maybe a -> a
Data.Maybe.fromMaybe :: a -> Maybe a -> a
Data.Maybe.isJust :: Maybe a -> Bool
Data.Maybe.isNothing :: Maybe a -> Bool
Data.Maybe.listToMaybe :: [a] -> Maybe a
Data.Maybe.mapMaybe :: (a -> Maybe b) -> [a] -> [b]
Data.Maybe.maybeToList :: Maybe a -> [a]
-- imported via Prelude
Just :: a -> Maybe a
data Maybe a = Nothing | Just a
Nothing :: Maybe a
maybe :: b -> (a -> b) -> Maybe a -> b
```

  This output shows that, in the context of the current session (ie in the scope of `Prelude`), the first group of items from `Data.Maybe` are not in scope (although they are available in fully qualified form in the GHCi session - see *What's really in scope at the prompt?* (page 24)), whereas the second group of items are in scope (via `Prelude`) and are therefore available either unqualified, or with a `Prelude.` qualifier.

:cd ⟨dir⟩
Changes the current working directory to ⟨dir⟩. A "~" symbol at the beginning of ⟨dir⟩ will be replaced by the contents of the environment variable `HOME`. See also the *:show paths* (page 54) command for showing the current working directory.

Note: changing directories causes all currently loaded modules to be unloaded. This is because the search path is usually expressed using relative directories, and changing the search path in the middle of a session is not supported.

:cmd ⟨expr⟩
Executes ⟨expr⟩ as a computation of type `IO String`, and then executes the resulting string as a list of GHCi commands. Multiple commands are separated by newlines. The *:cmd* (page 45) command is useful with *:def* (page 46) and *:set stop* (page 53).

:complete ⟨type⟩ [⟨n⟩-][⟨m⟩] ⟨string-literal⟩
This command allows to request command completions from GHCi even when interacting over a pipe instead of a proper terminal and is designed for integrating GHCi's completion with text editors and IDEs.

When called, *:complete* (page 45) prints the ⟨n⟩th to ⟨m⟩th completion candidates for the partial input ⟨string-literal⟩ for the completion domain denoted by ⟨type⟩. Currently, only the `repl` domain is supported which denotes the kind of completion that would be provided interactively by GHCi at the input prompt.

If omitted, ⟨n⟩ and ⟨m⟩ default to the first or last available completion candidate respectively. If there are less candidates than requested via the range argument, ⟨n⟩ and ⟨m⟩ are implicitly capped to the number of available completion candidates.

The output of *:complete* (page 45) begins with a header line containing three space-delimited fields:

- An integer denoting the number `l` of printed completions,
- an integer denoting the total number of completions available, and finally
- a string literal denoting a common prefix to be added to the returned completion candidates.

The header line is followed by ⟨l⟩ lines each containing one completion candidate encoded as (quoted) string literal. Here are some example invocations showing the various cases:

```
ghci> :complete repl 0 ""
0 470 ""
ghci> :complete repl 5 "import For"
5 21 "import "
"Foreign"
"Foreign.C"
"Foreign.C.Error"
"Foreign.C.String"
"Foreign.C.Types"
ghci> :complete repl 5-10 "import For"
6 21 "import "
"Foreign.C.Types"
"Foreign.Concurrent"
"Foreign.ForeignPtr"
"Foreign.ForeignPtr.Safe"
"Foreign.ForeignPtr.Unsafe"
"Foreign.Marshal"
ghci> :complete repl 20- "import For"
2 21 "import "
"Foreign.StablePtr"
"Foreign.Storable"
ghci> :complete repl "map"
3 3 ""
"map"
"mapM"
"mapM_"
ghci> :complete repl 5-10 "map"
0 3 ""
```

`:continue` `[⟨ignoreCount⟩]`

Continue the current evaluation, when stopped at a breakpoint.

If an `⟨ignoreCount⟩` is specified, the program will ignore the current breakpoint for the next `⟨ignoreCount⟩` iterations. See command *`:ignore`* (page 49).

`:def``[!] ⟨name⟩ ⟨expr⟩`

`:def` (page 46) is used to define new commands, or macros, in GHCi. The command `:def ⟨name⟩ ⟨expr⟩` defines a new GHCi command `:name`, implemented by the Haskell expression ⟨expr⟩, which must have type `String -> IO String`. When `:name args` is typed at the prompt, GHCi will run the expression `(name args)`, take the resulting `String`, and feed it back into GHCi as a new sequence of commands. Separate commands in the result must be separated by "`\n`".

That's all a little confusing, so here's a few examples. To start with, here's a new GHCi command which doesn't take any arguments or produce any results, it just outputs the current date and time:

```
ghci> let date _ = Data.Time.getZonedTime >>= print >> return ""
ghci> :def date date
```

(continues on next page)

(continued from previous page)

```
ghci> :date
2017-04-10 12:34:56.93213581 UTC
```

Here's an example of a command that takes an argument. It's a re-implementation of *:cd* (page 45):

```
ghci> let mycd d = System.Directory.setCurrentDirectory d >> return ""
ghci> :def mycd mycd
ghci> :mycd ..
```

Or I could define a simple way to invoke "`ghc --make Main`" in the current directory:

```
ghci> :def make (\_ -> return ":! ghc --make Main")
```

We can define a command that reads GHCi input from a file. This might be useful for creating a set of bindings that we want to repeatedly load into the GHCi session:

```
ghci> :def . readFile
ghci> :. cmds.ghci
```

Notice that we named the command `:.`, by analogy with the "`.`" Unix shell command that does the same thing.

Typing `:def` on its own lists the currently-defined macros. Attempting to redefine an existing command name results in an error unless the `:def!` form is used, in which case the old command with that name is silently overwritten. However for builtin commands the old command can still be used by preceding the command name with a double colon (eg `::load`). It's not possible to redefine the commands `:{`, `:}` and `:!`.

:delete `* | ⟨num⟩ ...`
Delete one or more breakpoints by number (use *:show breaks* (page 53) to see the number of each breakpoint). The * form deletes all the breakpoints.

:disable `* | ⟨num⟩ ...`
Disable one or more breakpoints by number (use *:show breaks* (page 53) to see the number and state of each breakpoint). The * form disables all the breakpoints.

:doc `⟨name⟩`
(Experimental: This command will likely change significantly in GHC 8.8.)

Displays the documentation for the given name. Currently the command is restricted to displaying the documentation directly on the declaration in question, ignoring documentation for arguments, constructors etc.

:edit `⟨file⟩`
Opens an editor to edit the file ⟨file⟩, or the most recently loaded module if ⟨file⟩ is omitted. If there were errors during the last loading, the cursor will be positioned at the line of the first error. The editor to invoke is taken from the *VISUAL* (page 47) or `EDITOR` environment variables, or a default editor on your system if neither is not set. You can change the editor using *:set editor* (page 52).

VISUAL

Hidden

:enable `* | ⟨num⟩ ...`
Enable one or more disabled breakpoints by number (use *:show breaks* (page 53) to see the number and state of each breakpoint). The * form enables all the disabled breakpoints. Enabling a break point will reset its `ignore count` to 0. (See *:ignore* (page 49))

:force ⟨identifier⟩ ...
: Prints the value of ⟨identifier⟩ in the same way as *:print* (page 51). Unlike *:print* (page 51), *:force* (page 48) evaluates each thunk that it encounters while traversing the value. This may cause exceptions or infinite loops, or further breakpoints (which are ignored, but displayed).

:forward ⟨n⟩
: Move forward ⟨n⟩ steps in the history. ⟨n⟩ is one if omitted. See *Tracing and history* (page 38) for more about GHCi's debugging facilities. See also: *:trace* (page 54), *:history* (page 48), *:back* (page 44).

:help
:?
: Displays a list of the available commands.

:
: Repeat the previous command.

:history [num]
: Display the history of evaluation steps. With a number, displays that many steps (default: 20). For use with *:trace* (page 54); see *Tracing and history* (page 38). To set the number of history entries stored by GHCi, use the *-fghci-hist-size=⟨n⟩* (page 39) flag.

:info[!] ⟨name⟩
: Displays information about the given name(s). For example, if ⟨name⟩ is a class, then the class methods and their types will be printed; if ⟨name⟩ is a type constructor, then its definition will be printed; if ⟨name⟩ is a function, then its type will be printed. If ⟨name⟩ has been loaded from a source file, then GHCi will also display the location of its definition in the source.

For types and classes, GHCi also summarises instances that mention them. To avoid showing irrelevant information, an instance is shown only if (a) its head mentions ⟨name⟩, and (b) all the other things mentioned in the instance are in scope (either qualified or otherwise) as a result of a *:load* (page 50) or *:module* (page 51) commands.

The command `:info!` works in a similar fashion but it removes restriction (b), showing all instances that are in scope and mention ⟨name⟩ in their head.

:instances ⟨type⟩
: Displays all the class instances available to the argument ⟨type⟩. The command will match ⟨type⟩ with the first parameter of every instance and then check that all constraints are satisfiable.

When combined with *PartialTypeSignatures* (page 520), a user can insert wildcards into a query and learn the constraints required of each wildcard for ⟨type⟩ match with an instance.

The output is a listing of all matching instances, simplified and instantiated as much as possible.

For example:

```
> :instances Maybe (Maybe Int)
instance Eq (Maybe (Maybe Int)) -- Defined in ‘GHC.Maybe’
instance Ord (Maybe (Maybe Int)) -- Defined in ‘GHC.Maybe’
instance Show (Maybe (Maybe Int)) -- Defined in ‘GHC.Show’
instance Read (Maybe (Maybe Int)) -- Defined in ‘GHC.Read’

> :set -XPartialTypeSignatures -fno-warn-partial-type-signatures
```

(continues on next page)

(continued from previous page)

```
> :instances Maybe _
instance Eq _ => Eq (Maybe _) -- Defined in 'GHC.Maybe'
instance Semigroup _ => Monoid (Maybe _) -- Defined in 'GHC.Base'
instance Ord _ => Ord (Maybe _) -- Defined in 'GHC.Maybe'
instance Semigroup _ => Semigroup (Maybe _) -- Defined in 'GHC.Base'
instance Show _ => Show (Maybe _) -- Defined in 'GHC.Show'
instance Read _ => Read (Maybe _) -- Defined in 'GHC.Read'
```

Only instances which could potentially be used will be displayed in the results. Instances which require unsatisfiable constraints such as `TypeError` will not be included. In the following example, the instance for `A` is not shown because it cannot be used.

```
ghci>:set -XDataKinds -XUndecidableInstances
ghci>import GHC.TypeLits
ghci>class A a
ghci>instance (TypeError (Text "Not possible")) => A Bool
ghci>:instances Bool
instance Eq Bool -- Defined in 'GHC.Classes'
instance Ord Bool -- Defined in 'GHC.Classes'
instance Enum Bool -- Defined in 'GHC.Enum'
instance Show Bool -- Defined in 'GHC.Show'
instance Read Bool -- Defined in 'GHC.Read'
instance Bounded Bool -- Defined in 'GHC.Enum'
```

:issafe `[⟨module⟩]`
Displays Safe Haskell information about the given module (or the current module if omitted). This includes the trust type of the module and its containing package.

:ignore `⟨break⟩ ⟨ignoreCount⟩`
Set the ignore count of the breakpoint with number `⟨break⟩` to `⟨ignoreCount⟩`.

The next `⟨ignoreCount⟩` times the program hits the breakpoint `⟨break⟩`, this breakpoint is ignored and the program doesn't stop. Every time the breakpoint is ignored, the `ignore count` is decremented by 1. When the `ignore count` is zero, the program again stops at the break point.

You can also specify an `⟨ignoreCount⟩` on a *:continue* (page 46) command when you resume execution of your program.

:kind`[!] ⟨type⟩`
Infers and prints the kind of ⟨type⟩. The latter can be an arbitrary type expression, including a partial application of a type constructor, such as `Either Int`. In fact, *:kind* (page 49) even allows you to write a partial application of a type synonym (usually disallowed), so that this works:

```
ghci> type T a b = (a,b,a)
ghci> :k T Int Bool
T Int Bool :: *
ghci> :k T
T :: * -> * -> *
ghci> :k T Int
T Int :: * -> *
```

Free type variables are also implicitly quantified, same as if you wrote `:k forall a. [a]` so this also works:

```
ghci> :k [a]
[a] :: *
```

If you specify the optional “!”, GHC will in addition normalise the type by expanding out type synonyms and evaluating type-function applications, and display the normalised result.

:list ⟨identifier⟩
Lists the source code around the definition of ⟨identifier⟩ or the current breakpoint if not given. This requires that the identifier be defined in an interpreted module. If your output device supports it, then GHCi will highlight the active subexpression in bold.

:list [⟨module⟩] ⟨line⟩
Lists the source code around the given line number of ⟨module⟩. This requires that the module be interpreted. If your output device supports it, then GHCi will highlight the active subexpression in bold.

:load[!] [*]⟨module⟩
Recursively loads the specified ⟨module⟩s, and all the modules they depend on. Here, each ⟨module⟩ must be a module name or filename, but may not be the name of a module in a package.

All previously loaded modules, except package modules, are forgotten. The new set of modules is known as the target set. Note that *:load* (page 50) can be used without any arguments to unload all the currently loaded modules and bindings.

Normally pre-compiled code for a module will be loaded if available, or otherwise the module will be compiled to byte-code. Using the * prefix forces a module to be loaded as byte-code.

Adding the optional “!” turns type errors into warnings while loading. This allows to use the portions of the module that are correct, even if there are type errors in some definitions. Effectively, the *-fdefer-type-errors* (page 414) flag is set before loading and unset after loading if the flag has not already been set before. See *Deferring type errors to runtime* (page 414) for further motivation and details.

After a *:load* (page 50) command, the current context is set to:

- ⟨module⟩, if it was loaded successfully, or
- the most recently successfully loaded module, if any other modules were loaded as a result of the current *:load* (page 50), or
- `Prelude` otherwise.

:loc-at ⟨module⟩ ⟨line⟩ ⟨col⟩ ⟨end-line⟩ ⟨end-col⟩ [⟨name⟩]
Tries to find the definition site of the name at the given source-code span, e.g.:

```
X> :loc-at X.hs 6 14 6 16 mu
X.hs:(8,7)-(8,9)
```

This command is useful when integrating GHCi with text editors and IDEs for providing a goto-definition facility.

The `:loc-at` command requires *:set +c* (page 56) to be set.

:main ⟨arg1⟩ ... ⟨argn⟩
When a program is compiled and executed, it can use the `getArgs` IO action to access the command-line arguments. However, we cannot simply pass the arguments to `main` while we are testing in ghci, as `main` doesn’t take its arguments directly.

Instead, we can use the *:main* (page 50) command. This runs whatever `main` is in scope, with any arguments being treated the same as command-line arguments, e.g.:

```
ghci> main = System.Environment.getArgs >>= print
ghci> :main foo bar
["foo","bar"]
```

We can also quote arguments which contains characters like spaces, and they are treated like Haskell strings, or we can just use Haskell list syntax:

```
ghci> :main foo "bar baz"
["foo","bar baz"]
ghci> :main ["foo", "bar baz"]
["foo","bar baz"]
```

Finally, other IO actions can be called, either with the `-main-is` flag or the *:run* (page 52) command:

```
ghci> foo = putStrLn "foo" >> System.Environment.getArgs >>= print
ghci> bar = putStrLn "bar" >> System.Environment.getArgs >>= print
ghci> :set -main-is foo
ghci> :main foo "bar baz"
foo
["foo","bar baz"]
ghci> :run bar ["foo", "bar baz"]
bar
["foo","bar baz"]
```

:module `+|- [*]⟨mod1⟩ ...`

import `⟨mod⟩`
Sets or modifies the current context for statements typed at the prompt. The form `import mod` is equivalent to `:module +mod`. See *What's really in scope at the prompt?* (page 24) for more details.

:print `⟨names⟩`
Prints a value without forcing its evaluation. *:print* (page 51) may be used on values whose types are unknown or partially known, which might be the case for local variables with polymorphic types at a breakpoint. While inspecting the runtime value, *:print* (page 51) attempts to reconstruct the type of the value, and will elaborate the type in GHCi's environment if possible. If any unevaluated components (thunks) are encountered, then *:print* (page 51) binds a fresh variable with a name beginning with `_t` to each thunk. See *Breakpoints and inspecting variables* (page 32) for more information. See also the *:sprint* (page 54) command, which works like *:print* (page 51) but does not bind new variables.

:quit
Quits GHCi. You can also quit by typing `Control-D` at the prompt.

:reload`[!]`
Attempts to reload the current target set (see *:load* (page 50)) if any of the modules in the set, or any dependent module, has changed. Note that this may entail loading new modules, or dropping modules which are no longer indirectly required by the target.

Adding the optional "`!`" turns type errors into warnings while loading. This allows to use the portions of the module that are correct, even if there are type errors in some definitions. Effectively, the *-fdefer-type-errors* (page 414) flag is set before loading and unset after loading if the flag has not already been set before. See *Deferring type errors to runtime* (page 414) for further motivation and details.

:run
See *:main* (page 50).

:script [⟨n⟩] ⟨filename⟩
Executes the lines of a file as a series of GHCi commands. The syntax for file-name arguments respects shell quoting rules, i.e., file names containing spaces can be enclosed in double quotes or with spaces escaped with a backslash. This command is compatible with multiline statements as set by *:set +m* (page 56)

:set [⟨option⟩ ...]
Sets various options. See *The :set and :seti commands* (page 56) for a list of available options and *Interactive-mode options* (page 143) for a list of GHCi-specific flags. The *:set* (page 52) command by itself shows which options are currently set. It also lists the current dynamic flag settings, with GHCi-specific flags listed separately.

:set args ⟨arg⟩
Sets the list of arguments which are returned when the program calls `System.Environment.getArgs`.

:set editor ⟨cmd⟩
Sets the command used by *:edit* (page 47) to ⟨cmd⟩.

:set local-config ⟨source|ignore⟩
If `ignore`, `./.ghci` files will be ignored (sourcing untrusted local scripts is a security risk). The default is `source`. Set this directive in your user `.ghci` script, i.e. before the local script would be sourced.

Even when set to `ignore`, a local script will still be processed if given by *-ghci-script* (page 59) on the command line, or sourced via *:script* (page 52).

:set prog ⟨prog⟩
Sets the string to be returned when the program calls `System.Environment.getProgName`.

:set prompt ⟨prompt⟩
Sets the string to be used as the prompt in GHCi. Inside ⟨prompt⟩, the next sequences are replaced:

- `%s` by the names of the modules currently in scope.
- `%l` by the line number (as referenced in compiler messages) of the current prompt.
- `%d` by the date in "Weekday Month Date" format (e.g., "Tue May 26") .
- `%t` by the current time in 24-hour HH:MM:SS format.
- `%T` by the current time in 12-hour HH:MM:SS format.
- `%@` by the current time in 12-hour am/pm format.
- `%A` by the current time in 24-hour HH:MM format.
- `%u` by the username of the current user.
- `%w` by the current working directory.
- `%o` by the operating system.
- `%a` by the machine architecture.
- `%N` by the compiler name.
- `%V` by the compiler version.
- `%call(cmd [args])` by the result of calling `cmd args`.

- `%%` by `%`.

If ⟨prompt⟩ starts with `"` then it is parsed as a Haskell String; otherwise it is treated as a literal string.

:set prompt-cont ⟨prompt⟩
Sets the string to be used as the continuation prompt (used when using the *:{* (page 21) command) in GHCi.

:set prompt-function ⟨prompt-function⟩
Sets the function to be used for the prompt displaying in GHCi. The function should be of the type `[String] -> Int -> IO String`. This function is called each time the prompt is being made. The first argument stands for the names of the modules currently in scope (the name of the "topmost" module will begin with a *; see *What's really in scope at the prompt?* (page 24) for more information). The second arguments is the line number (as referenced in compiler messages) of the current prompt.

:set prompt-cont-function ⟨prompt-function⟩
Sets the function to be used for the continuation prompt (used when using the *:{* (page 21) command) displaying in GHCi.

:set stop ⟨num⟩ ⟨cmd⟩
Set a command to be executed when a breakpoint is hit, or a new item in the history is selected. The most common use of *:set stop* (page 53) is to display the source code at the current location, e.g. `:set stop :list`.

If a number is given before the command, then the commands are run when the specified breakpoint (only) is hit. This can be quite useful: for example, `:set stop 1 :continue` effectively disables breakpoint 1, by running *:continue* (page 46) whenever it is hit In this case GHCi will still emit a message to say the breakpoint was hit. If you don't want such a message, you can use the *:disable* (page 47) command. What's more, with cunning use of *:def* (page 46) and *:cmd* (page 45) you can use *:set stop* (page 53) to implement conditional breakpoints:

```
*ghci> :def cond \expr -> return (":cmd if (" ++ expr ++ ") then return \"\" else␣
↪return \":continue\"")
*ghci> :set stop 0 :cond (x < 3)
```

To ignore breakpoints for a specified number of iterations use the *:ignore* (page 49) or the ⟨ignoreCount⟩ parameter of the *:continue* (page 46) command.

:seti [⟨option⟩ ...]
Like *:set* (page 52), but options set with *:seti* (page 53) affect only expressions and commands typed at the prompt, and not modules loaded with *:load* (page 50) (in contrast, options set with *:set* (page 52) apply everywhere). See *Setting options for interactive evaluation only* (page 57).

Without any arguments, displays the current set of options that are applied to expressions and commands typed at the prompt.

:show bindings
Show the bindings made at the prompt and their types.

:show breaks
List the active breakpoints.

:show context
List the active evaluations that are stopped at breakpoints.

:show imports
Show the imports that are currently in force, as created by `import` and *:module* (page 51) commands.

:show modules
Show the list of modules currently loaded.

:show packages
Show the currently active package flags, as well as the list of packages currently loaded.

:show paths
Show the current working directory (as set via *:cd* (page 45) command), as well as the list of directories searched for source files (as set by the `-i` option).

:show language
Show the currently active language flags for source files.

:showi language
Show the currently active language flags for expressions typed at the prompt (see also *:seti* (page 53)).

:show targets
Show list of currently loaded modules. This set of loaded modules can be modified by *:load* (page 50), *:reload* (page 51), *:add* (page 44) and *:unadd* (page 55).

:show `[args|prog|prompt|editor|stop]`
Displays the specified setting (see *:set* (page 52)).

:sprint `⟨expr⟩`
Prints a value without forcing its evaluation. *:sprint* (page 54) is similar to *:print* (page 51), with the difference that unevaluated subterms are not bound to new variables, they are simply denoted by _.

:step `[⟨expr⟩]`
Enable all breakpoints and begin evaluating an expression in single-stepping mode. In this mode evaluation will be stopped after every reduction, allowing local variables to be inspected. If ⟨expr⟩ is not given, evaluation will resume at the last breakpoint. See *Single-stepping* (page 37).

:steplocal
Enable only breakpoints in the current top-level binding and resume evaluation at the last breakpoint. Continuation with *:steplocal* (page 54) is not possible if this last breakpoint was hit by an error (*-fbreak-on-error* (page 40)) or an exception (*-fbreak-on-exception* (page 40)).

:stepmodule
Enable only breakpoints in the current module and resume evaluation at the last breakpoint.

:trace `⟨expr⟩`
Evaluates the given expression (or from the last breakpoint if no expression is given), and additionally logs the evaluation steps for later inspection using *:history* (page 48). See *Tracing and history* (page 38).

:type `⟨expression⟩`
Infers and prints the type of ⟨expression⟩, solving constraints and reducing type families as much as possible. For polymorphic types, it does not instantiate any forall quantified variables.

```
*X> :type length
length :: Foldable t => t a -> Int
```

Type family reduction is skipped if the function is not fully instantiated, as this has been observed to give more intuitive results. You may want to use *:info* (page 48) if you are not applying any arguments, as that will return the original type of the function.

:type +d ⟨expression⟩
Infers and prints the type of ⟨expression⟩, instantiating *all* the forall quantifiers, solving constraints, defaulting, and generalising. In this mode, if the inferred type is constrained by any interactive class (`Num`, `Show`, `Eq`, `Ord`, `Foldable`, or `Traversable`), the constrained type variable(s) are defaulted according to the rules described under *ExtendedDefaultRules* (page 29). This mode is quite useful when the inferred type is quite general (such as for `foldr`) and it may be helpful to see a more concrete instantiation.

```
*X> :type +d length
length :: [a] -> Int
```

:type-at ⟨path⟩ ⟨line⟩ ⟨col⟩ ⟨end-line⟩ ⟨end-col⟩ [⟨name⟩]
Reports the inferred type at the given span/position in the module, e.g.:

```
*X> :type-at X.hs 6 6 6 7 f
Int -> Int
```

This command is useful when integrating GHCi with text editors and IDEs for providing a show-type-under-point facility.

The first parameter (path) must be a file path and not a module name. The type of this path is dependent on how the module was loaded into GHCi: If the module was loaded by name, then the path name calculated by GHCi as described in *Modules vs. filenames* (page 17) must be used. If the module was loaded with an absolute or a relative path, then the same path must be specified.

The last string parameter is useful for when the span is out of date, i.e. the file changed and the code has moved. In which case *:type-at* (page 55) falls back to a general *:type* (page 54) like lookup.

The *:type-at* (page 55) command requires *:set +c* (page 56) to be set.

:unadd ⟨module⟩
Removes ⟨module⟩(s) from the current target set, and perform a reload (see *:add* (page 44) above).

:undef ⟨name⟩
Undefines the user-defined command ⟨name⟩ (see *:def* (page 46) above).

:unset ⟨option⟩
Unsets certain options. See *The :set and :seti commands* (page 56) for a list of available options.

:uses ⟨module⟩ ⟨line⟩ ⟨col⟩ ⟨end-line⟩ ⟨end-col⟩ [⟨name⟩]
Reports all module-local uses of the thing at the given position in the module, e.g.:

```
:uses GhciFind.hs 53 66 53 70 name
GhciFind.hs:(46,25)-(46,29)
GhciFind.hs:(47,37)-(47,41)
```

(continues on next page)

(continued from previous page)

```
GhciFind.hs:(53,66)-(53,70)
GhciFind.hs:(57,62)-(57,66)
```

This command is useful for highlighting and navigating all uses of an identifier in editors and IDEs.

The *:uses* (page 55) command requires *:set +c* (page 56) to be set.

:: ⟨builtin-command⟩
Executes the GHCi built-in command (e.g. `::type 3`). That is, look up on the list of builtin commands, excluding defined macros. See also: *:def* (page 46).

:! ⟨command⟩
Executes the shell command ⟨command⟩.

3.8 The :set and :seti commands

The *:set* (page 52) command sets two types of options: GHCi options, which begin with "+", and "command-line" options, which begin with "-".

Note: At the moment, the *:set* (page 52) command doesn't support any kind of quoting in its arguments: quotes will not be removed and cannot be used to group words together. For example, `:set -DFOO='BAR BAZ'` will not do what you expect.

3.8.1 GHCi options

GHCi options may be set using *:set* (page 52) and unset using *:unset* (page 55).

The available GHCi options are:

:set +c
Collect type and location information after loading modules. The commands *:all-types* (page 44), *:loc-at* (page 50), *:type-at* (page 55), and *:uses* (page 55) require `+c` to be active.

:set +m
Enable parsing of multiline commands. A multiline command is prompted for when the current input line contains open layout contexts (see *Multiline input* (page 22)).

:set +r
Normally, any evaluation of top-level expressions (otherwise known as CAFs or Constant Applicative Forms) in loaded modules is retained between evaluations. Turning on `+r` causes all evaluation of top-level expressions to be discarded after each evaluation (they are still retained *during* a single evaluation).

This option may help if the evaluated top-level expressions are consuming large amounts of space, or if you need repeatable performance measurements.

:set +s
Display some stats after evaluating each expression, including the elapsed time and number of bytes allocated. NOTE: the allocation figure is only accurate to the size of the storage manager's allocation area, because it is calculated at every GC. Hence, you might see values of zero if no GC has occurred.

:set +t

Display the type of each variable bound after a statement is entered at the prompt. If the statement is a single expression, then the only variable binding will be for the variable `it`.

3.8.2 Setting GHC command-line options in GHCi

Normal GHC command-line options may also be set using *:set* (page 52). For example, to turn on *-Wmissing-signatures* (page 97), you would say:

```
ghci> :set -Wmissing-signatures
```

GHCi will also accept any file-header pragmas it finds, such as `{-# OPTIONS_GHC ... #-}` and `{-# LANGUAGE ... #-}` (see *Pragmas* (page 604)). For example, instead of using *:set* (page 52) to enable *-Wmissing-signatures* (page 97), you could instead write:

```
ghci> {-# OPTIONS_GHC -Wmissing-signatures #-}
```

Any GHC command-line option that is designated as dynamic (see the table in *Flag reference* (page 134)), may be set using *:set* (page 52). To unset an option, you can set the reverse option:

```
ghci> :set -Wno-incomplete-patterns -XNoMultiParamTypeClasses
```

Flag reference (page 134) lists the reverse for each option where applicable.

Certain static options (*-package ⟨pkg⟩* (page 221), *-I⟨dir⟩* (page 240), *-i⟨dir⟩[:⟨dir⟩]** (page 201), and *-l ⟨lib⟩* (page 245) in particular) will also work, but some may not take effect until the next reload.

3.8.3 Setting options for interactive evaluation only

GHCi actually maintains *two* sets of options:

- The *loading options* apply when loading modules
- The *interactive options* apply when evaluating expressions and commands typed at the GHCi prompt.

The *:set* (page 52) command modifies both, but there is also a *:seti* (page 53) command (for "set interactive") that affects only the interactive options set.

It is often useful to change the interactive options, without having that option apply to loaded modules too. For example

```
:seti -XMonoLocalBinds
```

It would be undesirable if *MonoLocalBinds* (page 526) were to apply to loaded modules too: that might cause a compilation error, but more commonly it will cause extra recompilation, because GHC will think that it needs to recompile the module because the flags have changed.

If you are setting language options in your `.ghci` file, it is good practice to use *:seti* (page 53) rather than *:set* (page 52), unless you really do want them to apply to all modules you load in GHCi.

The two sets of options can be inspected using the *:set* (page 52) and *:seti* (page 53) commands respectively, with no arguments. For example, in a clean GHCi session we might see something like this:

```
ghci> :seti
base language is: GHC2021
with the following modifiers:
  -XExtendedDefaultRules
  -XNoMonomorphismRestriction
GHCi-specific dynamic flag settings:
other dynamic, non-language, flag settings:
  -fexternal-dynamic-refs
  -fignore-optim-changes
  -fignore-hpc-changes
  -fimplicit-import-qualified
warning settings:
```

The two sets of options are initialised as follows. First, both sets of options are initialised as described in *The .ghci and .haskeline files* (page 58). Then the interactive options are modified as follows:

- The option `-XExtendedDefaultRules` is enabled, in order to apply special defaulting rules to expressions typed at the prompt (see *Type defaulting in GHCi* (page 29)).
- The Monomorphism Restriction is disabled (see *Switching off the Monomorphism Restriction* (page 525)).

3.9 The `.ghci` and `.haskeline` files

3.9.1 The `.ghci` files

When it starts, unless the *-ignore-dot-ghci* (page 59) flag is given, GHCi reads and executes commands from the following files, in this order, if they exist:

1. *ghcappdata*/`ghci.conf`, where ⟨ghcappdata⟩ depends on your system, but is usually something like `$HOME/.ghc` or `$XDG_CONFIG_HOME/ghc` on Unix or `C:\Users{username}\AppData\Roaming\ghc` on Windows.
2. `./.ghci`

The `ghci.conf` file is most useful for turning on favourite options (e.g. `:set +s`), and defining useful macros.

Note: When setting language options in this file it is usually desirable to use *:seti* (page 53) rather than *:set* (page 52) (see *Setting options for interactive evaluation only* (page 57)).

Placing a `.ghci` file in a directory with a Haskell project is a useful way to set certain project-wide options so you don't have to type them every time you start GHCi: eg. if your project uses multi-parameter type classes, scoped type variables, and CPP, and has source files in three subdirectories A, B and C, you might put the following lines in `.ghci`:

```
:set -XMultiParamTypeClasses -XScopedTypeVariables -cpp
:set -iA:B:C
```

(Note that strictly speaking the *-i* (page 201) flag is a static one, but in fact it works to set it using *:set* (page 52) like this. The changes won't take effect until the next *:load* (page 50), though.)

> **Warning:** Sourcing untrusted `./.ghci` files is a security risk. They can contain arbitrary commands that will be executed as the user. Use *:set local-config* (page 52) to inhibit the processing of `./.ghci` files.

Once you have a library of GHCi macros, you may want to source them from separate files, or you may want to source your `.ghci` file into your running GHCi session while debugging it

```
:def source readFile
```

With this macro defined in your `.ghci` file, you can use `:source file` to read GHCi commands from `file`. You can find (and contribute!-) other suggestions for `.ghci` files on this Haskell wiki page: GHC/GHCi

Additionally, any files specified with *-ghci-script* (page 59) flags will be read after the standard files, allowing the use of custom .ghci files.

Two command-line options control whether the startup files files are read:

-ignore-dot-ghci
: Don't read either `./.ghci` or the other startup files when starting up.

-ghci-script
: Read a specific file after the usual startup files. May be specified repeatedly for multiple inputs. *-ignore-dot-ghci* (page 59) does not apply to these files.

When defining GHCi macros, there is some important behavior you should be aware of when names may conflict with built-in commands, especially regarding tab completion.

For example, consider if you had a macro named `:time` and in the shell, typed `:t 3` — what should happen? The current algorithm we use for completing commands is:

1. First, look up an exact match on the name from the defined macros.
2. Look for the exact match on the name in the built-in command list.
3. Do a prefix lookup on the list of built-in commands - if a built-in command matches, but a macro is defined with the same name as the built-in defined, pick the macro.
4. Do a prefix lookup on the list of built-in commands.
5. Do a prefix lookup on the list of defined macros.

Here are some examples:

1. You have a macro `:time` and enter `:t 3`

   You get `:type 3`

2. You have a macro `:type` and enter `:t 3`

   You get `:type 3` with your defined macro, not the builtin.

3. You have a macro `:time` and a macro `:type`, and enter `:t 3`

   You get `:type 3` with your defined macro.

When giving priority to built-in commands, you can use *:: (builtin-command)* (page 56), like `::type 3`.

3.9.2 The `.haskeline` file

GHCi uses Haskeline under the hood. You can configure it to, among other things, prune duplicates from GHCi history. See: Haskeline user preferences.

3.10 Compiling to object code inside GHCi

By default, GHCi compiles Haskell source code into byte-code that is interpreted by the runtime system. GHCi can also compile Haskell code to object code: to turn on this feature, use the *`-fobject-code`* (page 244) flag either on the command line or with *`:set`* (page 52) (the option *`-fbyte-code`* (page 244) restores byte-code compilation again). Compiling to object code takes longer, but typically the code will execute 10-20 times faster than byte-code.

Compiling to object code inside GHCi is particularly useful if you are developing a compiled application, because the *`:reload`* (page 51) command typically runs much faster than restarting GHC with *`--make`* (page 68) from the command-line, because all the interface files are already cached in memory.

There are disadvantages to compiling to object-code: you can't set breakpoints in object-code modules, for example. Only the exports of an object-code module will be visible in GHCi, rather than all top-level bindings as in interpreted modules.

3.11 Running the interpreter in a separate process

Normally GHCi runs the interpreted code in the same process as GHC itself, on top of the same RTS and sharing the same heap. However, if the flag *`-fexternal-interpreter`* (page 60) is given, then GHC will spawn a separate process for running interpreted code, and communicate with it using messages over a pipe.

`-fexternal-interpreter`

Since 8.0.1

Run interpreted code (for GHCi, Template Haskell, Quasi-quoting, or Annotations) in a separate process. The interpreter will run in profiling mode if *`-prof`* (page 656) is in effect, and in dynamically-linked mode if *`-dynamic`* (page 246) is in effect.

There are a couple of caveats that will hopefully be removed in the future: this option is currently not implemented on Windows (it is a no-op), and the external interpreter does not support the GHCi debugger, so breakpoints and single-stepping don't work with *`-fexternal-interpreter`* (page 60).

See also the *`-pgmi ⟨cmd⟩`* (page 238) (*Replacing the program for one or more phases* (page 238)) and *`-opti ⟨option⟩`* (page 239) (*Forcing options to a particular phase* (page 239)) flags.

Why might we want to do this? The main reason is that the RTS running the interpreted code can be a different flavour (profiling or dynamically-linked) from GHC itself. So for example:

- We can use the profiler to collect stack traces when using GHCi (see *Stack Traces in GHCi* (page 31)).
- When compiling Template Haskell code with *`-prof`* (page 656) we don't need to compile the modules without *`-prof`* (page 656) first (see *Using Template Haskell with Profiling* (page 537)) because we can run the profiled object code in the interpreter.

This feature is experimental in GHC 8.0.x, but it may become the default in future releases.

3.12 Running the interpreter on a different host

When using the flag *-fexternal-interpreter* (page 60) GHC will spawn and communicate with the separate process using pipes. There are scenarios (e.g. when cross compiling) where it is favourable to have the communication happen over the network. GHC provides two utilities for this, which can be found in the `utils` directory.

- `remote-iserv` needs to be built with the cross compiler to be executed on the remote host. Or in the case of using it on the same host the stage2 compiler will do as well.
- `iserv-proxy` needs to be built on the build machine by the build compiler.

After starting `remote-iserv ⟨tmp_dir⟩ ⟨port⟩` on the target and providing it with a temporary folder (where it will copy the necessary libraries to load to) and port it will listen for the proxy to connect.

Providing *-pgmi ⟨/path/to/iserv-proxy⟩* (page 238) and *-opti ⟨slave-ip⟩ -opti ⟨slave-port⟩ [-opti -v]* (page 239) in addition to *-fexternal-interpreter* (page 60) will then make ghc go through the proxy instead.

There are some limitations when using this. File and process IO will be executed on the target. As such packages like `git-embed`, `file-embed` and others might not behave as expected if the target and host do not share the same filesystem.

3.13 Building GHCi libraries

When invoked in the static way, GHCi will use the GHC RTS's static runtime linker to load object files for imported modules when available. However, when these modules are built with *-split-sections* (page 246) this linking can be quite expensive. To reduce this cost, package managers and build systems may opt to produce a pre-linked *GHCi object* using the *--merge-objs* (page 68) mode. This merges the per-module objects into a single object, collapsing function sections into a single text section which can be efficiently loaded by the runtime linker.

3.14 FAQ and Things To Watch Out For

The interpreter can't load modules with foreign export declarations! Unfortunately not. We haven't implemented it yet. Please compile any offending modules by hand before loading them into GHCi.

-O (page 113) is ineffective in GHCi!

Before GHC 9.8, optimizations were considered too unstable to be used with the bytecode interpreter. This restriction has been lifted, but is still regarded as experimental and guarded by *-funoptimized-core-for-interpreter* (page 267), which is enabled by default. In order to use optimizations, run:

```
ghci -fno-unoptimized-core-for-interpreter -O
```

Concurrent threads don't carry on running when GHCi is waiting for input. This should work, as long as your GHCi was built with the *-threaded* (page 248) switch, which is the default. Consult whoever supplied your GHCi installation.

After using `getContents`, I can't use `stdin`, until I do `:load` or `:reload` This is the defined behaviour of `getContents`: it puts the stdin Handle in a state known as semi-closed, wherein any further I/O operations on it are forbidden. Because I/O state is retained between computations, the semi-closed state persists until the next *:load* (page 50) or *:reload* (page 51) command.

You can make `stdin` reset itself after every evaluation by giving GHCi the command `:set +r`. This works because `stdin` is just a top-level expression that can be reverted to its unevaluated state in the same way as any other top-level expression (CAF).

I can't use `Control-C` to interrupt computations in GHCi on Windows. See *Running GHCi on Windows* (page 703).

The default buffering mode is different in GHCi to GHC. In GHC, the stdout handle is line-buffered by default. However, in GHCi we turn off the buffering on stdout, because this is normally what you want in an interpreter: output appears as it is generated.

If you want line-buffered behaviour, as in GHC, you can start your program thus:

```
main = do { hSetBuffering stdout LineBuffering; ... }
```

CHAPTER

FOUR

USING RUNGHC

runghc/runhaskell allows you to run Haskell programs using the interpreter, instead of having to compile them first.

4.1 Usage

The runghc command-line looks like:

```
runghc [runghc flags] [GHC flags] module [program args]
```

Any flags not recognized by runghc are automatically passed to GHC. If a flag is recognized by both runghc and GHC but you want to pass it to GHC then you can place it after a -- separator. Flags after the separator are treated as GHC only flags. Alternatively you can use the runghc option --ghc-arg=<arg> to pass any flag or argument directly to GHC.

module could be a Haskell source filename with or without the extension. If for some reason the filename starts with a - you can use a second -- to indicate the end of flags. Anything following a second -- will be considered a program file or module name followed by its arguments. For example:

- runghc -- -- -hello.hs

4.2 runghc flags

runghc accepts the following flags:

- -f /path/to/ghc: tell runghc the path of GHC executable to use to run the program. By default runghc will search for GHC in the directories in the system search path.
- --ghc-arg=<arg>: Pass an option or argument to GHC
- --help: print usage information.
- --version: print version information.

4.3 GHC Flags

As discussed earlier, use `--` or `--ghc-arg=<arg>` to disambiguate GHC flags when needed. For example, `-f` is recognized by runghc, therefore to pass `-fliberate-case` to GHC use any of the following:

- `runghc -- -fliberate-case`
- `runghc --ghc-arg=-fliberate-case`

Note that any non-flag arguments are never passed to GHC. An unused non-flag argument will be considered as the name of the program to run. If a GHC flag takes an argument use `--ghc-arg=<arg>` to pass the argument to GHC. For example, if you want to pass `-package foo` to GHC use any of the following:

- `runghc -package --ghc-arg=foo Main.hs`
- `runghc --ghc-arg=-package --ghc-arg=foo Main.hs`

CHAPTER

FIVE

USING GHC

5.1 Using GHC

5.1.1 Getting started: compiling programs

In this chapter you'll find a complete reference to the GHC command-line syntax, including all 400+ flags. It's a large and complex system, and there are lots of details, so it can be quite hard to figure out how to get started. With that in mind, this introductory section provides a quick introduction to the basic usage of GHC for compiling a Haskell program, before the following sections dive into the full syntax.

Let's create a Hello World program, and compile and run it. First, create a file `hello.hs` containing the Haskell code:

```
main = putStrLn "Hello, World!"
```

To compile the program, use GHC like this:

```
$ ghc hello.hs
```

(where `$` represents the prompt: don't type it). GHC will compile the source file `hello.hs`, producing an object file `hello.o` and an interface file `hello.hi`, and then it will link the object file to the libraries that come with GHC to produce an executable called `hello` on Unix/Linux/Mac, or `hello.exe` on Windows.

By default GHC will be very quiet about what it is doing, only printing error messages. If you want to see in more detail what's going on behind the scenes, add `-v` (page 76) to the command line.

Then we can run the program like this:

```
$ ./hello
Hello World!
```

If your program contains multiple modules, then you only need to tell GHC the name of the source file containing the `Main` module, and GHC will examine the `import` declarations to find the other modules that make up the program and find their source files. This means that, with the exception of the `Main` module, every source file should be named after the module name that it contains (with dots replaced by directory separators). For example, the module `Data.Person` would be in the file `Data/Person.hs` on Unix/Linux/Mac, or `Data\Person.hs` on Windows.

5.1.2 Options overview

GHC's behaviour is controlled by options, which for historical reasons are also sometimes referred to as command-line flags or arguments. Options can be specified in three ways:

5.1.2.1 Command-line arguments

An invocation of GHC takes the following form:

```
ghc [argument...]
```

Command-line arguments are either options or file names.

Command-line options begin with `-`. They may *not* be grouped: `-v0` is different from `-v -0`. Options need not precede filenames: e.g., `ghc *.o -o foo`. All options are processed and then applied to all files; you cannot, for example, invoke `ghc -c -O1 Foo.hs -O2 Bar.hs` to apply different optimisation levels to the files `Foo.hs` and `Bar.hs`.

In addition to passing arguments via the command-line, arguments can be passed via GNU-style response files. For instance,

```
$ cat response-file
-O1
Hello.hs
-o Hello
$ ghc @response-file
```

Note: Note that command-line options are *order-dependent*, with arguments being evaluated from left-to-right. This can have seemingly strange effects in the presence of flag implication. For instance, consider *-fno-specialise* (page 124) and *-O1* (page 113) (which implies *-fspecialise* (page 124)). These two command lines mean very different things:

`-fno-specialise -O1`

`-fspecialise` will be enabled as the `-fno-specialise` is overridden by the `-O1`.

`-O1 -fno-specialise`

`-fspecialise` will not be enabled, since the `-fno-specialise` overrides the `-fspecialise` implied by `-O1`.

5.1.2.2 Command line options in source files

Sometimes it is useful to make the connection between a source file and the command-line options it requires quite tight. For instance, if a Haskell source file deliberately uses name shadowing, it should be compiled with the `-Wno-name-shadowing` option. Rather than maintaining the list of per-file options in a `Makefile`, it is possible to do this directly in the source file using the `OPTIONS_GHC` *pragma* (page 605)

```
{-# OPTIONS_GHC -Wno-name-shadowing #-}
module X where
...
```

`OPTIONS_GHC` is a *file-header pragma* (see *OPTIONS_GHC pragma* (page 605)).

Only *dynamic* flags can be used in an `OPTIONS_GHC` pragma (see *Dynamic and Mode options* (page 67)).

Note that your command shell does not get to the source file options, they are just included literally in the array of command-line arguments the compiler maintains internally, so you'll be desperately disappointed if you try to glob etc. inside `OPTIONS_GHC`.

Note: The contents of `OPTIONS_GHC` are appended to the command-line options, so options given in the source file override those given on the command-line.

It is not recommended to move all the contents of your Makefiles into your source files, but in some circumstances, the `OPTIONS_GHC` pragma is the Right Thing. (If you use *-keep-hc-file* (page 204) and have `OPTION` flags in your module, the `OPTIONS_GHC` will get put into the generated `.hc` file).

5.1.2.3 Setting options in GHCi

Options may also be modified from within GHCi, using the *:set* (page 52) command.

5.1.3 Dynamic and Mode options

Each of GHC's command line options is classified as dynamic or mode:

> Mode: A mode may be used on the command line only. You can pass only one mode flag. For example, *--make* (page 68) or *-E* (page 68). The available modes are listed in *Modes of operation* (page 68).
>
> Dynamic: A dynamic flag may be used on the command line, in a `OPTIONS_GHC` pragma in a source file, or set using *:set* (page 52) in GHCi.

The flag reference tables (*Flag reference* (page 134)) lists the status of each flag.

5.1.4 Meaningful file suffixes

File names with "meaningful" suffixes (e.g., `.lhs` or `.o`) cause the "right thing" to happen to those files.

.hs A Haskell module.

.lhs A "literate Haskell" module.

.hspp A file created by the preprocessor.

.hi A Haskell interface file, probably compiler-generated.

.hie An extended Haskell interface file, produced by the Haskell compiler.

.hc Intermediate C file produced by the Haskell compiler.

.c A C file not produced by the Haskell compiler.

.ll An llvm-intermediate-language source file, usually produced by the compiler.

.bc An llvm-intermediate-language bitcode file, usually produced by the compiler.

.s An assembly-language source file, usually produced by the compiler.

.o An object file, produced by an assembler.

Files with other suffixes (or without suffixes) are passed straight to the linker.

5.1.5 Modes of operation

GHC's behaviour is firstly controlled by a mode flag. Only one of these flags may be given, but it does not necessarily need to be the first option on the command-line. For instance,

```
$ ghc Main.hs --make -o my-application
```

If no mode flag is present, then GHC will enter *--make* (page 68) mode (*Using ghc --make* (page 71)) if there are any Haskell source files given on the command line, or else it will link the objects named on the command line to produce an executable.

The available mode flags are:

--interactive
: Interactive mode, which is also available as **ghci**. Interactive mode is described in more detail in *Using GHCi* (page 15).

--run ⟨file⟩
: Run a script's `main` entry-point. Similar to `runghc/runhaskell` this will by default use the bytecode interpreter. If the command-line contains a `--` argument then all arguments that follow will be passed to the script. All arguments that precede `--` are interpreted as GHC arguments.

--make
: In this mode, GHC will build a multi-module Haskell program automatically, figuring out dependencies for itself. If you have a straightforward Haskell program, this is likely to be much easier, and faster, than using **make**. Make mode is described in *Using ghc --make* (page 71).

  This mode is the default if there are any Haskell source files mentioned on the command line, and in this case the *--make* (page 68) option can be omitted.

-e ⟨expr⟩
: Expression-evaluation mode. This is very similar to interactive mode, except that there is a single expression to evaluate (⟨expr⟩) which is given on the command line. This flag may be given multiple times, in which case each expression is evaluated sequentially. See *Expression evaluation mode* (page 74) for more details.

-E
: Stop after preprocessing (`.hspp` file)

-C
: Stop after generating C (`.hc` file)

-S
: Stop after generating assembly (`.s` file)

-c
: Stop after generating object (`.o`) file

  This is the traditional batch-compiler mode, in which GHC can compile source files one at a time, or link objects together into an executable. See *Batch compiler mode* (page 75).

--merge-objs
: Merge a set of static object files into a library optimised for loading in GHCi. See *Building GHCi libraries* (page 61).

-M
Dependency-generation mode. In this mode, GHC can be used to generate dependency information suitable for use in a `Makefile`. See *Dependency generation* (page 216).

--frontend ⟨module⟩
Run GHC using the given frontend plugin. See *Frontend plugins* (page 645) for details.

-shared
Create a shared object (or, on Windows, DLL). See *Creating a DLL* (page 706).

--help
-?
Cause GHC to spew a long usage message to standard output and then exit.

--show-iface ⟨file⟩
Read the interface in ⟨file⟩ and dump it as text to `stdout`. For example `ghc --show-iface M.hi`.

--supported-extensions
--supported-languages
Print the supported language extensions.

--show-options
Print the supported command line options. This flag can be used for autocompletion in a shell.

--info
Print information about the compiler.

--version
-V
Print a one-line string including GHC's version number.

--numeric-version
Print GHC's numeric version number only.

--print-booter-version
Print the numeric version of the GHC binary used to bootstrap the build of this compiler.

--print-build-platform
Print the target string of the build platform, on which GHC was built, as generated by GNU Autotools. The format is `cpu-manufacturer-operating_system-(kernel)`, e.g., `x86_64-unknown-linux`.

--print-c-compiler-flags
List the flags passed to the C compiler during GHC build.

--print-c-compiler-link-flags
List the flags passed to the C compiler for the linking step during GHC build.

--print-debug-on
Print `True` if GHC was built with `-DDebug` flag. This enables assertions and extra debug code. The flag can be set in `GhcStage1HcOpts` and/or `GhcStage2HcOpts` and is automatically set for `devel1` and `devel2` build flavors.

--print-global-package-db
Print the path to GHC's global package database directory. A package database stores details about installed packages as a directory containing a file for each package. This flag prints the path to the global database shipped with GHC, and looks something like `/usr/lib/ghc/package.conf.d` on Unix. There may be other package databases, e.g., the user package databse. For more details see *Package Databases* (page 224).

--print-have-interpreter
Print YES if GHC was compiled to include the interpreter, NO otherwise. If this GHC does not have the interpreter included, running it in interactive mode (see *--interactive* (page 68)) will throw an error. This only pertains the use of GHC interactively, not any separate GHCi binaries (see *Using GHCi* (page 15)).

--print-have-native-code-generator
Print YES if native code generator supports the target platform, NO otherwise. (See *Native Code Generator (-fasm)* (page 236))

--print-host-platform
Print the target string of the host platform, i.e., the one on which GHC is supposed to run, as generated by GNU Autotools. The format is `cpu-manufacturer-operating_system-(kernel)`, e.g., `x86_64-unknown-linux`.

--print-leading-underscore
Print YES if GHC was compiled to use symbols with leading underscores in object files, NO otherwise. This is usually atarget platform dependent.

--print-libdir
Print the path to GHC's library directory. This is the top of the directory tree containing GHC's libraries, interfaces, and include files (usually something like `/usr/local/lib/ghc-5.04` on Unix). This is the value of `$libdir` in the package configuration file (see *Packages* (page 219)).

--print-ld-flags
Print linke flags used to compile GHC.

--print-object-splitting-supported
Prints NO as object splitting is no longer supported. See *-split-sections* (page 246) for a more portable and reliable alternative.

--print-project-git-commit-id
Print the Git commit id from which this GHC was built. This can be used to trace the current binary back to a specific revision, which is especially useful during development on GHC itself. It is set by the configure script.

--print-project-version
Print the version set in the configure script during build. This is simply the GHC version.

--print-rts-ways
Packages, like the Runtime System, can be built in a number of ways: - profiling - with profiling support - dynamic - with dynamic linking - logging - RTS event logging - threaded - mulithreaded RTS - debug - RTS with debug information

Various combinations of these flavours are possible.

--print-stage
GHC is built using GHC itself and this build happens in stages, which are numbered.

- Stage 0 is the GHC you have installed. The "GHC you have installed" is also called "the bootstrap compiler".
- Stage 1 is the first GHC we build, using stage 0. Stage 1 is then used to build the packages.
- Stage 2 is the second GHC we build, using stage 1. This is the one we normally install when you say make install.
- Stage 3 is optional, but is sometimes built to test stage 2.

Stage 1 does not support interactive execution (GHCi) and Template Haskell.

--print-support-smp
Print YES if GHC was built with multiporcessor support, NO otherwise.

--print-tables-next-to-code
Print YES if GHC was built with the flag `--enable-tables-next-to-code`, NO otherwise. This option is on by default, as it generates a more efficient code layout.

--print-target-platform
Print the target string of the target platform, i.e., the one on which generated binaries will run, as generated by GNU Autotools. The format is `cpu-manufacturer-operating_system-(kernel)`, e.g., `x86_64-unknown-linux`.

--print-unregisterised
Print YES if this GHC was built in unregisterised mode, NO otherwise. "Unregisterised" means that GHC will disable most platform-specific tricks and optimisations. Only the LLVM and C code generators will be available. See *Unregisterised compilation* (page 237) for more details.

5.1.5.1 Using `ghc --make`

In this mode, GHC will build a multi-module Haskell program by following dependencies from one or more root modules (usually just `Main`). For example, if your `Main` module is in a file called `Main.hs`, you could compile and link the program like this:

```
ghc --make Main.hs
```

In fact, GHC enters make mode automatically if there are any Haskell source files on the command line and no other mode is specified, so in this case we could just type

```
ghc Main.hs
```

Any number of source file names or module names may be specified; GHC will figure out all the modules in the program by following the imports from these initial modules. It will then attempt to compile each module which is out of date, and finally, if there is a `Main` module, the program will also be linked into an executable.

The main advantages to using `ghc --make` over traditional `Makefile`s are:

- GHC doesn't have to be restarted for each compilation, which means it can cache information between compilations. Compiling a multi-module program with `ghc --make` can be up to twice as fast as running `ghc` individually on each source file.
- You don't have to write a `Makefile`.
- GHC re-calculates the dependencies each time it is invoked, so the dependencies never get out of sync with the source.
- Using the *-j[⟨n⟩]* (page 72) flag, you can compile modules in parallel. Specify `-j ⟨n⟩` to compile ⟨n⟩ jobs in parallel. If ⟨n⟩ is omitted, then it defaults to the number of processors.

Any of the command-line options described in the rest of this chapter can be used with `--make`, but note that any options you give on the command line will apply to all the source files compiled, so if you want any options to apply to a single source file only, you'll need to use an `OPTIONS_GHC` pragma (see *Command line options in source files* (page 66)).

If the program needs to be linked with additional objects (say, some auxiliary C code), then the object files can be given on the command line and GHC will include them when linking the executable.

For backward compatibility with existing make scripts, when used in combination with *-c* (page 68), the linking phase is omitted (same as `--make -no-link`).

Note that GHC can only follow dependencies if it has the source file available, so if your program includes a module for which there is no source file, even if you have an object and an interface file for the module, then GHC will complain. The exception to this rule is for package modules, which may or may not have source files.

The source files for the program don't all need to be in the same directory; the *-i* (page 201) option can be used to add directories to the search path (see *The search path* (page 201)).

-j[⟨n⟩]
: Perform compilation in parallel when possible. GHC will use up to ⟨N⟩ threads during compilation. If N is omitted, then it defaults to the number of processors. Note that compilation of a module may not begin until its dependencies have been built.

5.1.5.2 GHC Jobserver Protocol

The GHC Jobserver Protocol was specified in GHC proposal #540.

This protocol allows a server to dynamically invoke many instances of a client process, while restricting all of those instances to use no more than <n> capabilities. This is achieved by coordination over a system semaphore (either a POSIX semaphore in the case of Linux and Darwin, or a Win32 semaphore in the case of Windows platforms).

There are two kinds of participants in the GHC Jobserver protocol:

- The *jobserver* creates a system semaphore with a certain number of available tokens.

  Each time the jobserver wants to spawn a new jobclient subprocess, it **must** first acquire a single token from the semaphore, before spawning the subprocess. This token **must** be released once the subprocess terminates.

  Once work is finished, the jobserver **must** destroy the semaphore it created.

- A *jobclient* is a subprocess spawned by the jobserver or another jobclient.

  Each jobclient starts with one available token (its *implicit token*, which was acquired by the parent which spawned it), and can request more tokens through the Jobserver Protocol by waiting on the semaphore.

  Each time a jobclient wants to spawn a new jobclient subprocess, it **must** pass on a single token to the child jobclient. This token can either be the jobclient's implicit token, or another token which the jobclient acquired from the semaphore.

  Each jobclient **must** release exactly as many tokens as it has acquired from the semaphore (this does not include the implicit tokens).

  GHC itself acts as a jobclient which can be enabled by using the flag `-jsem`.

-jsem
: Perform compilation in parallel when possible, coordinating with other processes through the semaphore ⟨sem⟩ (specified as a string). Error if the semaphore doesn't exist.

  Use of `-jsem` will override use of :ghc-flag:`-j[⟨n⟩]`, and vice-versa.

5.1.5.3 Multiple Home Units

The compiler also has support for building multiple units in a single compiler invocation. In modern projects it is common to work on multiple interdependent packages at once, using the support for multiple home units you can load all these local packages into one ghc session and quickly get feedback about how changes affect other dependent packages.

In order to specify multiple units, the *-unit @⟨filename⟩* (page 73) is given multiple times with a response file containing the arguments for each unit. The response file contains a newline separated list of arguments.

```
ghc -unit @unitA -unit @unitB
```

where the `unitA` response file contains the normal arguments that you would pass to `--make` mode.

```
-this-unit-id a-0.1.0.0
-i
-isrc
A1
A2
...
```

Then when the compiler starts in `--make` mode it will compile both units `a` and `b`.

There is also very basic support for multiple home units in GHCi, at the moment you can start a GHCi session with multiple units but only the *:reload* (page 51) is supported.

-unit @⟨filename⟩
    This option is passed multiple times to inform the compiler about all the home units which it will compile. The options for each unit are supplied in a response file which contains a newline separated list of normal arguments.

There are a few extra flags which have been introduced to make working with multiple units easier.

-working-dir ⟨dir⟩
    It is common to assume that a package is compiled in the directory where its cabal file resides. Thus, all paths used in the compiler are assumed to be relative to this directory. When there are multiple home units the compiler is often not operating in the standard directory and instead where the cabal.project file is located. In this case the *-working-dir* option can be passed which specifies the path from the current directory to the directory the unit assumes to be its root, normally the directory which contains the cabal file.

    When the flag is passed, any relative paths used by the compiler are offset by the working directory. Notably this includes *-i* (page 201) and *-I⟨dir⟩* (page 240) flags.

    This option can also be queried by the `getPackageRoot` Template Haskell function. It is intended to be used with helper functions such as `makeRelativeToProject` which make relative filepaths relative to the compilation directory rather than the directory which contains the .cabal file.

-this-package-name ⟨unit-id⟩
    This flag papers over the awkward interaction of the *PackageImports* (page 319) and multiple home units. When using `PackageImports` you can specify the name of the package in an import to disambiguate between modules which appear in multiple packages with the same name.

This flag allows a home unit to be given a package name so that you can also disambiguate between multiple home units which provide modules with the same name.

-hidden-module ⟨module name⟩
This flag can be supplied multiple times in order to specify which modules in a home unit should not be visible outside of the unit it belongs to.

The main use of this flag is to be able to recreate the difference between an exposed and hidden module for installed packages.

-reexported-module ⟨module name⟩
This flag can be supplied multiple times in order to specify which modules are not defined in a unit but should be reexported. The effect is that other units will see this module as if it was defined in this unit.

The use of this flag is to be able to replicate the reexported modules feature of packages with multiple home units.

The home unit closure requirement

There is one very important closure property which you must ensure when using multiple home units.

> Any external unit must not depend on any home unit.

This closure property is checked by the compiler but it's up to the tool invoking GHC to ensure that the supplied list of home units obeys this invariant.

For example, if we have three units, `p`, `q` and `r`, where `p` depends on `q` and `q` depends on `r`, then the closure property states that if we load `p` and `r` as home units then we must also load `q`, because `q` depends on the home unit `r` and we need `q` because `p` depends on it.

5.1.5.4 Expression evaluation mode

This mode is very similar to interactive mode, except that there is a single expression to evaluate which is specified on the command line as an argument to the `-e` option:

```
ghc -e expr
```

Haskell source files may be named on the command line, and they will be loaded exactly as in interactive mode. The expression is evaluated in the context of the loaded modules.

For example, to load and run a Haskell program containing a module `Main`, we might say:

```
ghc -e Main.main Main.hs
```

or we can just use this mode to evaluate expressions in the context of the `Prelude`:

```
$ ghc -e "interact (unlines.map reverse.lines)"
hello
olleh
```

5.1.5.5 Batch compiler mode

In *batch mode*, GHC will compile one or more source files given on the command line.

The first phase to run is determined by each input-file suffix, and the last phase is determined by a flag. If no relevant flag is present, then go all the way through to linking. This table summarises:

Phase of the compilation system	Suffix saying "start here"	Flag saying "stop after"	(suffix of) output file
literate pre-processor	`.lhs`		`.hs`
C pre-processor (opt.)	`.hs` (with `-cpp`)	`-E`	`.hspp`
Haskell compiler	`.hs`	`-C`, `-S`	`.hc`, `.s`
C compiler (opt.)	`.hc` or `.c`	`-S`	`.s`
assembler	`.s`	`-c`	`.o`
linker	⟨other⟩		`a.out`

Thus, a common invocation would be:

```
ghc -c Foo.hs
```

to compile the Haskell source file `Foo.hs` to an object file `Foo.o`.

Note: What the Haskell compiler proper produces depends on what backend code generator is used. See *GHC Backends* (page 236) for more details.

Note: Pre-processing is optional, the *-cpp* (page 240) flag turns it on. See *Options affecting the C pre-processor* (page 240) for more details.

Note: The option *-E* (page 68) runs just the pre-processing passes of the compiler, dumping the result in a file.

Note: The option *-C* (page 68) is only available when GHC is built in unregisterised mode. See *Unregisterised compilation* (page 237) for more details.

Overriding the default behaviour for a file

As described above, the way in which a file is processed by GHC depends on its suffix. This behaviour can be overridden using the *-x ⟨suffix⟩* (page 75) option:

-x ⟨suffix⟩
: Causes all files following this option on the command line to be processed as if they had the suffix ⟨suffix⟩. For example, to compile a Haskell module in the file `M.my-hs`, use `ghc -c -x hs M.my-hs`.

5.1.6 Verbosity options

See also the `--help`, `--version`, `--numeric-version`, and `--print-libdir` modes in *Modes of operation* (page 68).

`-v`
The *`-v`* (page 76) option makes GHC *verbose*: it reports its version number and shows (on stderr) exactly how it invokes each phase of the compilation system. Moreover, it passes the `-v` flag to most phases; each reports its version number (and possibly some other information).

Please, oh please, use the `-v` option when reporting bugs! Knowing that you ran the right bits in the right order is always the first thing we want to verify.

`-v⟨n⟩`
To provide more control over the compiler's verbosity, the `-v` flag takes an optional numeric argument. Specifying `-v` on its own is equivalent to `-v3`, and the other levels have the following meanings:

`-v0` Disable all non-essential messages (this is the default).

`-v1` Minimal verbosity: print one line per compilation (this is the default when *`--make`* (page 68) or *`--interactive`* (page 68) is on).

`-v2` Print the name of each compilation phase as it is executed. (equivalent to *`-dshow-passes`* (page 256)).

`-v3` The same as `-v2`, except that in addition the full command line (if appropriate) for each compilation phase is also printed.

`-v4` The same as `-v3` except that the intermediate program representation after each compilation phase is also printed (excluding preprocessed and C/assembly files).

`-fprint-potential-instances`
When GHC can't find an instance for a class, it displays a short list of some of the instances it knows about. With this flag it prints *all* the instances it knows about.

`-fhide-source-paths`
Starting with minimal verbosity (`-v1`, see *`-v`* (page 76)), GHC displays the name, the source path and the target path of each compiled module. This flag can be used to reduce GHC's output by hiding source paths and target paths.

The following flags control the way in which GHC displays types in error messages and in GHCi:

`-fprint-unicode-syntax`
When enabled GHC prints type signatures using the unicode symbols from the *`UnicodeSyntax`* (page 277) extension. For instance,

```
ghci> :set -fprint-unicode-syntax
ghci> :t +v (>>)
(>>) ∷ Monad m ⇒ ∀ a b. m a → m b → m b
```

`-fprint-explicit-foralls`
Using *`-fprint-explicit-foralls`* (page 76) makes GHC print explicit `forall` quantification at the top level of a type; normally this is suppressed. For example, in GHCi:

```
ghci> let f x = x
ghci> :t f
f :: a -> a
```

(continues on next page)

(continued from previous page)

```
ghci> :set -fprint-explicit-foralls
ghci> :t f
f :: forall a. a -> a
```

However, regardless of the flag setting, the quantifiers are printed under these circumstances:

- For nested `foralls`, e.g.

```
ghci> :t GHC.ST.runST
GHC.ST.runST :: (forall s. GHC.ST.ST s a) -> a
```

- If any of the quantified type variables has a kind that mentions a kind variable, e.g.

```
ghci> :i Data.Type.Equality.sym
Data.Type.Equality.sym ::
  forall k (a :: k) (b :: k).
  (a Data.Type.Equality.:~: b) -> b Data.Type.Equality.:~: a
        -- Defined in Data.Type.Equality
```

`-fprint-explicit-kinds`

Using *`-fprint-explicit-kinds`* (page 77) makes GHC print kind arguments in types, which are normally suppressed. This can be important when you are using kind polymorphism. For example:

```
ghci> :set -XPolyKinds
ghci> data T a (b :: l) = MkT
ghci> :t MkT
MkT :: forall k l (a :: k) (b :: l). T a b
ghci> :set -fprint-explicit-kinds
ghci> :t MkT
MkT :: forall k l (a :: k) (b :: l). T @{k} @l a b
ghci> :set -XNoPolyKinds
ghci> :t MkT
MkT :: T @{*} @* a b
```

In the output above, observe that `T` has two kind variables (`k` and `l`) and two type variables (`a` and `b`). Note that `k` is an *inferred* variable and `l` is a *specified* variable (see *Inferred vs. specified type variables* (page 387)), so as a result, they are displayed using slightly different syntax in the type `T @{k} @l a b`. The application of `l` (with `@l`) is the standard syntax for visible type application (see *Visible type application* (page 386)). The application of `k` (with `@{k}`), however, uses a hypothetical syntax for visible type application of inferred type variables. This syntax is not currently exposed to the programmer, but it is nevertheless displayed when *`-fprint-explicit-kinds`* (page 77) is enabled.

`-fprint-explicit-coercions`

Using *`-fprint-explicit-coercions`* (page 77) makes GHC print coercions in types. When trying to prove the equality between types of different kinds, GHC uses type-level coercions. Users will rarely need to see these, as they are meant to be internal.

`-fprint-axiom-incomps`

Using *`-fprint-axiom-incomps`* (page 77) tells GHC to display incompatibilities between closed type families' equations, whenever they are printed by *`:info`* (page 48) or *`--show-iface ⟨file⟩`* (page 69).

```
ghci> :i Data.Type.Equality.==
type family (==) (a :: k) (b :: k) :: Bool
  where
      (==) (f a) (g b) = (f == g) && (a == b)
      (==) a a = 'True
      (==) _1 _2 = 'False
ghci> :set -fprint-axiom-incomps
ghci> :i Data.Type.Equality.==
type family (==) (a :: k) (b :: k) :: Bool
  where
      {- #0 -} (==) (f a) (g b) = (f == g) && (a == b)
      {- #1 -} (==) a a = 'True
          -- incompatible with: #0
      {- #2 -} (==) _1 _2 = 'False
          -- incompatible with: #1, #0
```

The equations are numbered starting from 0, and the comment after each equation refers to all preceding equations it is incompatible with.

-fprint-equality-relations
Using *-fprint-equality-relations* (page 78) tells GHC to distinguish between its equality relations when printing. For example, ~ is homogeneous lifted equality (the kinds of its arguments are the same) while ~~ is heterogeneous lifted equality (the kinds of its arguments might be different) and ~# is heterogeneous unlifted equality, the internal equality relation used in GHC's solver. Generally, users should not need to worry about the subtleties here; ~ is probably what you want. Without *-fprint-equality-relations* (page 78), GHC prints all of these as ~. See also *Equality constraints* (page 499).

-fprint-expanded-synonyms
When enabled, GHC also prints type-synonym-expanded types in type errors. For example, with this type synonyms:

```
type Foo = Int
type Bar = Bool
type MyBarST s = ST s Bar
```

This error message:

```
Couldn't match type 'Int' with 'Bool'
Expected type: ST s Foo
  Actual type: MyBarST s
```

Becomes this:

```
Couldn't match type 'Int' with 'Bool'
Expected type: ST s Foo
  Actual type: MyBarST s
Type synonyms expanded:
Expected type: ST s Int
  Actual type: ST s Bool
```

-fprint-redundant-promotion-ticks
The *DataKinds* (page 357) extension allows us to use data constructors at the type level:

```
type B = True     -- refers to the data constructor True (of type Bool)
```

When there is a type constructor of the same name, it takes precedence during name resolution:

```
data True = MkT
type B = True     -- now refers to the type constructor (of kind Type)
```

We can tell GHC to prefer the data constructor over the type constructor using special namespace disambiguation syntax that we call a *promotion tick*:

```
data True = MkT
type B = 'True
    -- refers to the data constructor True (of type Bool)
    -- even in the presence of a type constructor of the same name
```

Note that the promotion tick is not a promotion operator. Its only purpose is to instruct GHC to prefer the promoted data constructor over a type constructor in case of a name conflict. Therefore, GHC will not print the tick when the name conflict is absent:

```
ghci> type B = False
ghci> :kind! B
B :: Bool
= False          -- no promotion tick here

ghci> data False -- introduce a name conflict

ghci> :kind! B
B :: Bool
= 'False         -- promotion tick resolves the name conflict
```

The *-fprint-redundant-promotion-ticks* (page 78) instructs GHC to print the promotion tick unconditionally.

-fprint-typechecker-elaboration

When enabled, GHC also prints extra information from the typechecker in warnings. For example:

```
main :: IO ()
main = do
  return $ let a = "hello" in a
  return ()
```

This warning message:

```
A do-notation statement discarded a result of type '[Char]'
Suppress this warning by saying
  '_ <- ($) return let a = "hello" in a'
or by using the flag -fno-warn-unused-do-bind
```

Becomes this:

```
A do-notation statement discarded a result of type '[Char]'
Suppress this warning by saying
  '_ <- ($)
          return
          let
            AbsBinds [] []
              {Exports: [a <= a
```

(continues on next page)

(continued from previous page)

```
                          <>]
                Exported types: a :: [Char]
                                [LclId, Str=DmdType]
                Binds: a = "hello"}
           in a'
or by using the flag -fno-warn-unused-do-bind
```

-fdefer-diagnostics

Causes GHC to group diagnostic messages by severity and output them after other messages when building a multi-module Haskell program. This flag can make diagnostic messages more visible when used in conjunction with *--make* (page 68) and *-j[⟨n⟩]* (page 72). Otherwise, it can be hard to find the relevant errors or likely to ignore the warnings when they are mixed with many other messages.

-fdiagnostics-as-json

Causes GHC to emit diagnostic messages in a standardized JSON format, and output them directly to `stderr`. The format follows the JSON Lines convention, where each diagnostic is its own JSON object separated by a new line.

The structure of the output is described by a JSON Schema. The schema can be downloaded `here`.

-fdiagnostics-color=⟨always|auto|never⟩

Causes GHC to display error messages with colors. To do this, the terminal must have support for ANSI color codes, or else garbled text will appear. The default value is `auto`, which means GHC will make an attempt to detect whether terminal supports colors and choose accordingly.

The precise color scheme is controlled by the environment variable `GHC_COLORS` (or `GHC_COLOURS`). This can be set to colon-separated list of `key=value` pairs. These are the default settings:

```
header=:message=1:warning=1;35:error=1;31:fatal=1;31:margin=1;34
```

Each value is expected to be a Select Graphic Rendition (SGR) substring. The formatting of each element can inherit from parent elements. For example, if `header` is left empty, it will inherit the formatting of `message`. Alternatively if `header` is set to `1` (bold), it will be bolded but still inherits the color of `message`.

Currently, in the primary message, the following inheritance tree is in place:

- `message`
  - `header`
    - `warning`
    - `error`
    - `fatal`

In the caret diagnostics, there is currently no inheritance at all between `margin`, `warning`, `error`, and `fatal`.

The environment variable can also be set to the magical values `never` or `always`, which is equivalent to setting the corresponding `-fdiagnostics-color` flag but with lower precedence.

-fdiagnostics-show-caret

Default on

Controls whether GHC displays a line of the original source code where the error was detected. This also affects the associated caret symbol that points at the region of code at fault.

-fshow-error-context

Default on

Controls whether GHC displays information about the context in which an error occurred. This controls whether the part of the error message which says "In the equation..", "In the pattern.." etc is displayed or not.

-fprint-error-index-links=⟨always|auto|never⟩

Default auto

Controls whether GHC will emit error indices as ANSI hyperlinks to the Haskell Error Index. When set to auto, this flag will render hyperlinks if the terminal is capable; when set to always, this flag will render the hyperlinks regardless of the capabilities of the terminal.

-ferror-spans

Causes GHC to emit the full source span of the syntactic entity relating to an error message. Normally, GHC emits the source location of the start of the syntactic entity only.

For example:

```
test.hs:3:6: parse error on input `where'
```

becomes:

```
test296.hs:3:6-10: parse error on input `where'
```

And multi-line spans are possible too:

```
test.hs:(5,4)-(6,7):
    Conflicting definitions for `a'
    Bound at: test.hs:5:4
              test.hs:6:7
    In the binding group for: a, b, a
```

Note that line numbers start counting at one, but column numbers start at zero. This choice was made to follow existing convention (i.e. this is how Emacs does it).

-fkeep-going

Since 8.10.1

Causes GHC to continue the compilation if a module has an error. Any reverse dependencies are pruned immediately and the whole compilation is still flagged as an error. This option has no effect if parallel compilation (*-j[⟨n⟩]* (page 72)) is in use.

-freverse-errors

Causes GHC to output errors in reverse line-number order, so that the errors and warnings that originate later in the file are displayed first.

-Rghc-timing

Prints a one-line summary of timing statistics for the GHC run. This option is equivalent to `+RTS -tstderr`, see *RTS options to control the garbage collector* (page 184).

5.1.7 Platform-specific Flags

Some flags only make sense for particular target platforms.

-mavx
(x86 only) These SIMD instructions are currently not supported by the *native code generator* (page 236). Enabling this flag has no effect and is only present for future extensions.

The *LLVM backend* (page 236) may use AVX if your processor supports it, but detects this automatically, so no flag is required.

-mavx2
(x86 only) These SIMD instructions are currently not supported by the *native code generator* (page 236). Enabling this flag has no effect and is only present for future extensions.

The *LLVM backend* (page 236) may use AVX2 if your processor supports it, but detects this automatically, so no flag is required.

-mavx512cd
(x86 only) These SIMD instructions are currently not supported by the *native code generator* (page 236). Enabling this flag has no effect and is only present for future extensions.

The *LLVM backend* (page 236) may use AVX512 if your processor supports it, but detects this automatically, so no flag is required.

-mavx512er
(x86 only) These SIMD instructions are currently not supported by the *native code generator* (page 236). Enabling this flag has no effect and is only present for future extensions.

The *LLVM backend* (page 236) may use AVX512 if your processor supports it, but detects this automatically, so no flag is required.

-mavx512f
(x86 only) These SIMD instructions are currently not supported by the *native code generator* (page 236). Enabling this flag has no effect and is only present for future extensions.

The *LLVM backend* (page 236) may use AVX512 if your processor supports it, but detects this automatically, so no flag is required.

-mavx512pf
(x86 only) These SIMD instructions are currently not supported by the *native code generator* (page 236). Enabling this flag has no effect and is only present for future extensions.

The *LLVM backend* (page 236) may use AVX512 if your processor supports it, but detects this automatically, so no flag is required.

-msse
(x86 only) Use the SSE registers and instruction set to implement floating point operations when using the *native code generator* (page 236). This gives a substantial performance improvement for floating point, but the resulting compiled code will only run on processors that support SSE (Intel Pentium 3 and later, or AMD Athlon XP and later). The *LLVM backend* (page 236) will also use SSE if your processor supports it but detects this automatically so no flag is required.

Since GHC 8.10, SSE2 is assumed to be present on both x86 and x86-64 platforms and will be used by default. Even when setting this flag, SSE2 will be used instead.

-msse2
(x86 only, added in GHC 7.0.1) Use the SSE2 registers and instruction set to implement floating point operations when using the *native code generator* (page 236). This gives a substantial performance improvement for floating point, but the resulting compiled

code will only run on processors that support SSE2 (Intel Pentium 4 and later, or AMD Athlon 64 and later). The *LLVM backend* (page 236) will also use SSE2 if your processor supports it but detects this automatically so no flag is required.

Since GHC 8.10, SSE2 is assumed to be present on both x86 and x86-64 platforms and will be used by default.

-msse3

(x86 only) Use the SSE3 instruction set to implement some floating point and bit operations when using the *native code generator* (page 236).

Note that the current version does not use SSE3 specific instructions and only requires SSE2 processor support.

The *LLVM backend* (page 236) will also use SSE3 if your processor supports it but detects this automatically so no flag is required.

-msse4

(x86 only) Use the SSE4 instruction set to implement some floating point and bit operations when using the *native code generator* (page 236).

Note that the current version does not use SSE4 specific instructions and only requires SSE2 processor support.

The *LLVM backend* (page 236) will also use SSE4 if your processor supports it but detects this automatically so no flag is required.

-msse4.2

(x86 only, added in GHC 7.4.1) Use the SSE4.2 instruction set to implement some floating point and bit operations when using the *native code generator* (page 236). The resulting compiled code will only run on processors that support SSE4.2 (Intel Core i7 and later). The *LLVM backend* (page 236) will also use SSE4.2 if your processor supports it but detects this automatically so no flag is required.

-mbmi

(x86 only) Use the BMI1 instruction set to implement some bit operations when using the *native code generator* (page 236).

Note that the current version does not use BMI specific instructions, so using this flag has no effect.

-mbmi2

(x86 only, added in GHC 7.4.1) Use the BMI2 instruction set to implement some bit operations when using the *native code generator* (page 236). The resulting compiled code will only run on processors that support BMI2 (Intel Haswell and newer, AMD Excavator, Zen and newer).

-mfma

Default off by default, except for Aarch64 where it's on by default.

Since 9.8.1

Use native FMA instructions to implement the fused multiply-add floating-point operations of the form `x * y + z`. This allows computing a multiplication and addition in a single instruction, without an intermediate rounding step. Supported architectures: X86 with the FMA3 instruction set (this includes most consumer processors since 2013), PowerPC and AArch64.

When this flag is disabled, GHC falls back to the C implementation of fused multiply-add, which might perform non-IEEE-compliant software emulation on some platforms (depending on the implementation of the C standard library).

5.1.8 Haddock

-haddock

By default, GHC ignores Haddock comments (`-- | ...` and `-- ^ ...`) and does not check that they're associated with a valid term, such as a top-level type-signature. With this flag GHC will parse Haddock comments and include them in the interface file it produces.

Consider using *-Winvalid-haddock* (page 107) to be informed about discarded documentation comments.

5.1.9 Miscellaneous flags

Some flags only make sense for a particular use case.

-ghcversion-file ⟨path to ghcversion.h⟩

When GHC is used to compile C files, GHC adds package include paths and includes `ghcversion.h` directly. The compiler will lookup the path for the `ghcversion.h` file from the `rts` package in the package database. In some cases, the compiler's package database does not contain the `rts` package, or one wants to specify a specific `ghcversions.h` to be included. This option can be used to specify the path to the `ghcversions.h` file to be included. This is primarily intended to be used by GHC's build system.

-H ⟨size⟩

Set the minimum size of the heap to ⟨size⟩. This option is equivalent to `+RTS -Hsize`, see *RTS options to control the garbage collector* (page 184).

5.1.9.1 Other environment variables

GHC can also be configured using various environment variables.

GHC_NO_UNICODE

When non-empty, disables Unicode diagnostics output regardless of locale settings. GHC can usually determine that locale is not Unicode-capable and fallback to ASCII automatically, but in some corner cases (e. g., when GHC output is redirected) you might hit `invalid argument (cannot encode character '\8216')`, in which case do set `GHC_NO_UNICODE`.

GHC_CHARENC

When set to `UTF-8` the compiler will always print UTF-8-encoded output, regardless of the current locale.

5.2 Warnings and sanity-checking

GHC has a number of options that select which types of non-fatal error messages, otherwise known as warnings, can be generated during compilation. Some options control individual warnings and others control collections of warnings. Use `-W⟨wflag⟩` to turn on an individual warning or a collection, or use `-Wno-⟨wflag⟩` to turn it off. Use `-Werror` to make all warnings into fatal errors, or `-Werror=⟨wflag⟩` to make a specific warning into an error. Reverse this with `-Wwarn` to make all warnings non-fatal, or `-Wwarn=⟨wflag⟩` to make a specific warning non-fatal.

Note: In GHC < 8 the syntax for `-W⟨wflag⟩` was `-fwarn-⟨wflag⟩` (e.g. `-fwarn-incomplete-patterns`). This spelling is deprecated, but still accepted for backwards compatibility. Likewise, `-Wno-⟨wflag⟩` used to be `fno-warn-⟨wflag⟩` (e.g. `-fno-warn-incomplete-patterns`).

5.2.1 Warning groups

The following flags are simple ways to select standard "packages" of warnings. They can be reversed using `-Wno-⟨group⟩`, which has the same effect as `-Wno-...` for every individual warning in the group.

`-Wdefault`

Since 8.0

By default, you get a standard set of warnings which are generally likely to indicate bugs in your program. These are:

- *`-Woverlapping-patterns`* (page 99)
- *`-Wwarnings-deprecations`* (page 90)
- *`-Wdeprecations`* (page 90)
- *`-Wdeprecated-flags`* (page 91)
- *`-Wunrecognised-pragmas`* (page 89)
- *`-Wduplicate-exports`* (page 93)
- *`-Wderiving-defaults`* (page 92)
- *`-Woverflowed-literals`* (page 92)
- *`-Wempty-enumerations`* (page 92)
- *`-Wmissing-fields`* (page 96)
- *`-Wmissing-methods`* (page 97)
- *`-Wwrong-do-bind`* (page 105)
- *`-Wsimplifiable-class-constraints`* (page 101)
- *`-Wtyped-holes`* (page 88)
- *`-Wdeferred-type-errors`* (page 88)
- *`-Wpartial-type-signatures`* (page 88)
- *`-Wunsupported-calling-conventions`* (page 91)
- *`-Wdodgy-foreign-imports`* (page 91)
- *`-Winline-rule-shadowing`* (page 106)
- *`-Wunsupported-llvm-version`* (page 101)
- *`-Wmissed-extra-shared-lib`* (page 101)
- *`-Wtabs`* (page 101)
- *`-Wunrecognised-warning-flags`* (page 88)
- *`-Winaccessible-code`* (page 100)
- *`-Wstar-binder`* (page 100)
- *`-Wstar-is-type`* (page 100)
- *`-Woperator-whitespace-ext-conflict`* (page 107)
- *`-Wambiguous-fields`* (page 108)
- *`-Wunicode-bidirectional-format-characters`* (page 108)
- *`-Wforall-identifier`* (page 108)
- *`-Wgadt-mono-local-binds`* (page 108)
- *`-Wtype-equality-requires-operators`* (page 109)
- *`-Wtype-equality-out-of-scope`* (page 109)
- *`-Wbadly-staged-types`* (page 111)
- *`-Winconsistent-flags`* (page 111)

- *-Wnoncanonical-monoid-instances* (page 91)
- *-Wnoncanonical-monad-instances* (page 90)
- *-Wdata-kinds-tc* (page 111)

-W

Provides the standard warnings plus

- *-Wunused-binds* (page 102)
- *-Wunused-matches* (page 103)
- *-Wunused-foralls* (page 104)
- *-Wunused-imports* (page 103)
- *-Wincomplete-patterns* (page 94)
- *-Wdodgy-exports* (page 92)
- *-Wdodgy-imports* (page 92)
- *-Wunbanged-strict-patterns* (page 106)

-Wextra

Alias for *-W* (page 86)

-Wall

Turns on all warning options that indicate potentially suspicious code. They include all warnings in *-Wextra* (page 86), plus:

- *-Whi-shadowing* (page 93)
- *-Wincomplete-record-updates* (page 95)
- *-Wincomplete-record-selectors* (page 95)
- *-Wincomplete-uni-patterns* (page 94)
- *-Wmissing-pattern-synonym-signatures* (page 97)
- *-Wmissing-signatures* (page 97)
- *-Wname-shadowing* (page 98)
- *-Worphans* (page 99)
- *-Wredundant-record-wildcards* (page 105)
- *-Wstar-is-type* (page 100)
- *-Wtrustworthy-safe* (page 587)
- *-Wtype-defaults* (page 101)
- *-Wunused-do-bind* (page 103)
- *-Wunused-record-wildcards* (page 104)
- *-Wincomplete-export-warnings* (page 111)

-Weverything

Since 8.0

Turns on every single warning supported by the compiler.

-Wcompat

Since 8.0

Turns on warnings that will be enabled by default in the future, but remain off in normal compilations for the time being. This allows library authors eager to make their code future compatible to adapt to new features before they even generate warnings.

This currently enables

- *-Wcompat-unqualified-imports* (page 88)
- *-Wimplicit-rhs-quantification* (page 110)
- *-Wdeprecated-type-abstractions* (page 110)

-w
Turns off all warnings, including the standard ones and those that *-Wall* (page 86) doesn't enable.

-Wnot
Deprecated alias for *-w* (page 86)

When a warning is emitted, the specific warning flag which controls it is shown, but the group can optionally be shown as well:

-fshow-warning-groups

Default off

When showing which flag controls a warning, also show the respective warning group flag(s) that warning is contained in.

5.2.2 Treating warnings as fatal errors

These options control which warnings are considered fatal and cause compilation to abort.

-Werror

Since 6.8 (-Wwarn)

Makes any warning into a fatal error. Useful so that you don't miss warnings when doing batch compilation. To reverse -Werror and stop treating any warnings as errors use -Wwarn, or use -Wwarn=⟨wflag⟩ to stop treating specific warnings as errors.

-Werror=⟨wflag⟩

Implies -W⟨wflag⟩

Makes a specific warning into a fatal error. The warning will be enabled if it hasn't been enabled yet. Can be reversed with -Wwarn=⟨wflag⟩.

-Werror=⟨group⟩ has the same effect as -Werror=... for each warning flag in the group (for example, -Werror=compat will turn every warning in the *-Wcompat* (page 86) group into a fatal error).

-Wwarn
Warnings are treated only as warnings, not as errors. This is the default, but can be useful to negate a *-Werror* (page 87) flag.

-Wwarn=⟨wflag⟩
Causes a specific warning to be treated as normal warning, not fatal error.

Note that it doesn't fully negate the effects of -Werror=⟨wflag⟩ - the warning will still be enabled.

-Wwarn=⟨group⟩ has the same effect as -Wwarn=... for each warning flag in the group (for example, -Wwarn=compat will mark every warning in the *-Wcompat* (page 86) group as non-fatal).

-Wno-error=⟨wflag⟩
Alternative spelling for -Wwarn=⟨wflag⟩.

5.2.3 Individual warning options

The full set of warning options is described below. To turn off any warning, simply give the corresponding `-Wno-...` option on the command line. For backwards compatibility with GHC versions prior to 8.0, all these warnings can still be controlled with `-f(no-)warn-*` instead of `-W(no-)*`.

`-Wunrecognised-warning-flags`

Since 8.0

Default on

Enables warnings when the compiler encounters a `-W...` flag that is not recognised.

`-Wcompat-unqualified-imports`

Since 8.10

Warns on unqualified imports of core library modules which are subject to change in future GHC releases. Currently the following modules are covered by this warning:

- `Data.List` due to the future addition of `Data.List.singleton` and specialisation of exports to the `[]` type. See the mailing list for details.

This warning can be addressed by either adding an explicit import list or using a `qualified` import.

`-Wprepositive-qualified-module`

Since 8.10

Normally, imports are qualified prepositively: `import qualified M`. By using *`ImportQualifiedPost`* (page 321), the qualified keyword can be used after the module name. Like so: `import M qualified`. This will warn when the first, prepositive syntax is used.

`-Wtyped-holes`

Since 7.8

Default on

Determines whether the compiler reports typed holes warnings. Has no effect unless typed holes errors are deferred until runtime. See *Typed Holes* (page 304) and *Deferring type errors to runtime* (page 414).

`-Wdeferred-type-errors`

Since 8.0

Default on

Causes a warning to be reported when a type error is deferred until runtime. See *Deferring type errors to runtime* (page 414).

`-Wdeferred-out-of-scope-variables`

Since 8.0

Warn when a deferred out-of-scope variable is encountered. See *Deferring type errors to runtime* (page 414).

`-Wpartial-type-signatures`

Since 7.10

Default on

Determines whether the compiler reports holes in partial type signatures as warnings. Has no effect unless *PartialTypeSignatures* (page 520) is enabled, which controls whether errors should be generated for holes in types or not. See *Partial Type Signatures* (page 520).

-fhelpful-errors

Since 7.4

Default on

When a name or package is not found in scope, make suggestions for the name or package you might have meant instead.

-Wunrecognised-pragmas

Since 6.10

Default on

Causes a warning to be emitted when a pragma that GHC doesn't recognise is used. As well as pragmas that GHC itself uses, GHC also recognises pragmas known to be used by other tools, e.g. `OPTIONS_HUGS` and `DERIVE`.

-Wmisplaced-pragmas

Since 9.4

Default on

Warn when a pragma that should only appear in the header of a module, such as a *LANGUAGE* or *OPTIONS_GHC* pragma, appears in the body of the module instead.

-Wmissed-specialisations

Since 8.0

Default off

Emits a warning if GHC cannot specialise an overloaded function, usually because the function needs an `INLINABLE` pragma. Reports when the situation arises during specialisation of an imported function.

This form is intended to catch cases where an imported function that is marked as `INLINABLE` (presumably to enable specialisation) cannot be specialised as it calls other functions that are themselves not specialised.

Note that this warning will not throw errors if used with *-Werror* (page 87).

-Wmissed-specializations
Alias for *-Wmissed-specialisations* (page 89)

-Wall-missed-specialisations

Since 8.0

Default off

Emits a warning if GHC cannot specialise an overloaded function, usually because the function needs an `INLINABLE` pragma. Reports all such situations.

Note that this warning will not throw errors if used with *-Werror* (page 87).

-Wall-missed-specializations
Alias for *-Wall-missed-specialisations* (page 89)

-Wextended-warnings

Since 9.8.1

Default on

Causes a warning to be emitted when a module, function or type with a `WARNING` or `DEPRECATED` pragma is used, regardless of the category which may be associated with the pragma. See *WARNING and DEPRECATED pragmas* (page 606) for more details on the pragmas. This implies *-Wdeprecations* (page 90) and all `-Wx-⟨category⟩` flags.

-Wx-⟨category⟩

Since 9.8.1

Default on

Causes a warning to be emitted when a module, function or type with a `WARNING in "x-⟨category⟩"` pragma is used. See *WARNING and DEPRECATED pragmas* (page 606) for more details on the pragmas.

-Wdeprecations

Default on

Causes a warning to be emitted when a module, function or type with `DEPRECATED pragma`, or a `WARNING` pragma with the `deprecated` category, is used. See *WARNING and DEPRECATED pragmas* (page 606) for more details on the pragmas.

-Wwarnings-deprecations

Since 6.10

Default on

Causes a warning to be emitted when a module, function or type with `DEPRECATED pragma`, or a `WARNING` pragma with the `deprecated` category, is used. See *WARNING and DEPRECATED pragmas* (page 606) for more details on the pragmas. An alias for *-Wdeprecations* (page 90).

-Wnoncanonical-monad-instances

Since 8.0

Default on

Warn if noncanonical `Applicative` or `Monad` instances declarations are detected.

When this warning is enabled, the following conditions are verified:

In `Monad` instances declarations warn if any of the following conditions does not hold:

- If `return` is defined it must be canonical (i.e. `return = pure`).
- If `(>>)` is defined it must be canonical (i.e. `(>>) = (*>)`).

Moreover, in `Applicative` instance declarations:

- Warn if `pure` is defined backwards (i.e. `pure = return`).
- Warn if `(*>)` is defined backwards (i.e. `(*>) = (>>)`).

-Wnoncanonical-monadfail-instances

Since 8.0

This warning is deprecated. It no longer has any effect since GHC 8.8. It was used during the transition period of the MonadFail proposal, to detect when an instance of the `Monad` class was not defined via `MonadFail`, or when a `MonadFail` instance was defined backwards, using the method in `Monad`.

-Wnoncanonical-monoid-instances

Since 8.0

Default on

Warn if noncanonical `Semigroup` or `Monoid` instances declarations are detected.

When this warning is enabled, the following conditions are verified:

In `Monoid` instances declarations warn if any of the following conditions does not hold:

- If `mappend` is defined it must be canonical (i.e. `mappend = (Data.Semigroup.<>)`).

Moreover, in `Semigroup` instance declarations:

- Warn if `(<>)` is defined backwards (i.e. `(<>) = mappend`).

-Wmissing-monadfail-instances

Since 8.0

This warning is deprecated. It no longer has any effect since GHC 8.8. It was used during the transition period of the MonadFail proposal, to warn when a failable pattern is used in a do-block that does not have a `MonadFail` instance.

-Wsemigroup

Since 8.0

This warning is deprecated. It no longer has any effect since GHC 9.8. It was used during the transition period of the semigroup proposal, to warn when an instance of `Monoid` was not an instance of `Semigroup`, or when a custom local operator `(<>)` could clash with *(<>)*, now exported from `Prelude`.

-Wdeprecated-flags

Since 6.10

Default on

Causes a warning to be emitted when a deprecated command-line flag is used.

-Wunsupported-calling-conventions

Since 7.6

Causes a warning to be emitted for foreign declarations that use unsupported calling conventions. In particular, if the `stdcall` calling convention is used on an architecture other than i386 then it will be treated as `ccall`.

-Wdodgy-foreign-imports

Since 6.10

Causes a warning to be emitted for foreign imports of the following form:

```
foreign import "f" f :: FunPtr t
```

on the grounds that it probably should be

```
foreign import "&f" f :: FunPtr t
```

The first form declares that f is a (pure) C function that takes no arguments and returns a pointer to a C function with type t, whereas the second form declares that f itself is a C function with type t. The first declaration is usually a mistake, and one that is hard to debug because it results in a crash, hence this warning.

-Wdodgy-exports

Since 6.12

Causes a warning to be emitted when a datatype T is exported with all constructors, i.e. T(..), but is it just a type synonym.

Also causes a warning to be emitted when a module is re-exported, but that module exports nothing.

-Wdodgy-imports

Since 6.8

Causes a warning to be emitted in the following cases:

- When a datatype T is imported with all constructors, i.e. T(..), but has been exported abstractly, i.e. T.
- When an import statement hides an entity that is not exported.

-Woverflowed-literals

Since 7.8

Causes a warning to be emitted if a literal will overflow, e.g. 300 :: Word8.

-Wempty-enumerations

Since 7.8

Causes a warning to be emitted if an enumeration is empty, e.g. [5 .. 3].

-Wderiving-defaults

Since 8.10

Causes a warning when both *DeriveAnyClass* (page 453) and *GeneralizedNewtypeDeriving* (page 447) are enabled and no explicit deriving strategy is in use. For example, this would result a warning:

```
class C a
newtype T a = MkT a deriving C
```

-Wduplicate-constraints

Since 7.8

Have the compiler warn about duplicate constraints in a type signature. For example

```
f :: (Eq a, Show a, Eq a) => a -> a
```

The warning will indicate the duplicated Eq a constraint.

This option is now deprecated in favour of *-Wredundant-constraints* (page 92).

-Wredundant-constraints

Since 8.0

Have the compiler warn about redundant constraints in a type signature. In particular:

- A redundant constraint within the type signature itself:

```
f :: (Eq a, Ord a) => a -> a
```

  The warning will indicate the redundant `Eq a` constraint: it is subsumed by the `Ord a` constraint.

- A constraint in the type signature is not used in the code it covers:

```
f :: Eq a => a -> a -> Bool
f x y = True
```

  The warning will indicate the redundant `Eq a` constraint: : it is not used by the definition of `f`.)

Similar warnings are given for a redundant constraint in an instance declaration.

When turning on, you can suppress it on a per-module basis with *-Wno-redundant-constraints* (page 92). Occasionally you may specifically want a function to have a more constrained signature than necessary, perhaps to leave yourself wiggle-room for changing the implementation without changing the API. In that case, you can suppress the warning on a per-function basis, using a call in a dead binding. For example:

```
f :: Eq a => a -> a -> Bool
f x y = True
where
    _ = x == x  -- Suppress the redundant-constraint warning for (Eq a)
```

Here the call to `(==)` makes GHC think that the `(Eq a)` constraint is needed, so no warning is issued.

-Wduplicate-exports

Since at least 5.04

Default on

Have the compiler warn about duplicate entries in export lists. This is useful information if you maintain large export lists, and want to avoid the continued export of a definition after you've deleted (one) mention of it in the export list.

-Whi-shadowing

Since at least 5.04, deprecated

Causes the compiler to emit a warning when a module or interface file in the current directory is shadowing one with the same module name in a library or other directory.

This flag was not implemented correctly and is now deprecated. It will be removed in a later version of GHC.

-Widentities

Since 7.2

Causes the compiler to emit a warning when a Prelude numeric conversion converts a type `T` to the same type `T`; such calls are probably no-ops and can be omitted. The functions checked for are: `toInteger`, `toRational`, `fromIntegral`, and `realToFrac`.

-Wimplicit-kind-vars

Since 8.6

This warning is deprecated. It no longer has any effect since GHC 8.10. It was used to detect if a kind variable is not explicitly quantified over. For instance, the following would produce a warning:

```
f :: forall (a :: k). Proxy a
```

This is now an error and can be fixed by explicitly quantifying over `k`:

```
f :: forall k (a :: k). Proxy a
```

or

```
f :: forall {k} (a :: k). Proxy a
```

-Wimplicit-lift

Since 9.2

Template Haskell quotes referring to local variables bound outside of the quote are implicitly converted to use `lift`. For example, `f x = [| reverse x |]` becomes `f x = [| reverse $(lift x) |]`. This flag issues a warning for every such implicit addition of `lift`. This can be useful when debugging more complex staged programs, where an implicit `lift` can accidentally conceal a variable used at a wrong stage.

-Wimplicit-prelude

Since 6.8

Default off

Have the compiler warn if the Prelude is implicitly imported. This happens unless either the Prelude module is explicitly imported with an `import ... Prelude ...` line, or this implicit import is disabled (either by *NoImplicitPrelude* (page 295) or a `LANGUAGE NoImplicitPrelude` pragma).

Note that no warning is given for syntax that implicitly refers to the Prelude, even if *NoImplicitPrelude* (page 295) would change whether it refers to the Prelude. For example, no warning is given when `368` means `Prelude.fromInteger (368::Prelude.Integer)` (where `Prelude` refers to the actual Prelude module, regardless of the imports of the module being compiled).

-Wincomplete-patterns

Since 5.04

The option *-Wincomplete-patterns* (page 94) warns about places where a pattern-match might fail at runtime. The function `g` below will fail when applied to non-empty lists, so the compiler will emit a warning about this when *-Wincomplete-patterns* (page 94) is enabled.

```
g [] = 2
```

This option isn't enabled by default because it can be a bit noisy, and it doesn't always indicate a bug in the program. However, it's generally considered good practice to cover all the cases in your functions, and it is switched on by *-W* (page 86).

-Wincomplete-uni-patterns

Since 7.2

The flag *-Wincomplete-uni-patterns* (page 94) is similar to *-Wincomplete-patterns* (page 94), except that it applies only to lambda-expressions and pattern bindings, constructs that only allow a single pattern:

```
h = \[] -> 2
Just k = f y
```

Furthermore, this flag also applies to lazy patterns, since they are syntactic sugar for pattern bindings. For example, `f ~(Just x) = (x,x)` is equivalent to `f y = let Just x = y in (x,x)`.

-fmax-pmcheck-models=⟨n⟩

Since 8.10

Default 30

The pattern match checker works by assigning symbolic values to each pattern. We call each such assignment a 'model'. Now, each pattern match clause leads to potentially multiple splits of that model, encoding different ways for the pattern match to fail. For example, when matching `x` against `Just 4`, we split each incoming matching model into two uncovered sub-models: One where `x` is `Nothing` and one where `x` is `Just y` but `y` is not 4.

This can be exponential in the arity of the pattern and in the number of guards in some cases. The *-fmax-pmcheck-models=⟨n⟩* (page 95) limit makes sure we scale polynomially in the number of patterns, by forgetting refined information gained from a partially successful match. For the above example, if we had a limit of 1, we would continue checking the next clause with the original, unrefined model.

-Wincomplete-record-updates

Since 6.4

The function `f` below will fail when applied to `Bar`, so the compiler will emit a warning about this when *-Wincomplete-record-updates* (page 95) is enabled.

```
data Foo = Foo { x :: Int }
         | Bar

f :: Foo -> Foo
f foo = foo { x = 6 }
```

This option isn't enabled by default because it can be very noisy, and it often doesn't indicate a bug in the program.

-Wincomplete-record-selectors

Since 9.10

When a record selector is applied to a constructor that does not contain that field, it will produce an error. For example

```
data T = T1 | T2 { x :: Int }

f :: T -> Int
f a = x a -- `f T1` will fail

g1 :: HasField "x" t Int => t -> Int
g1 a = 1 + getField @"x" a
```

(continues on next page)

(continued from previous page)

```
g2 :: T -> Int
g2 a = g1 a + 2 -- `g2 T1` will fail as well
```

The warning warns about cases like that. It also takes into account previously pattern-matched cases, for example

```
d :: T -> Int
d T1 = 0
d a = x a -- would not warn
```

-Wmissing-deriving-strategies

Since 8.8.1

Default off

The datatype below derives the `Eq` typeclass, but doesn't specify a strategy. When *-Wmissing-deriving-strategies* (page 96) is enabled, the compiler will emit a warning about this.

```
data Foo a = Foo a
  deriving (Eq)
```

The compiler will warn here that the deriving clause doesn't specify a strategy. If the warning is enabled, but *DerivingStrategies* (page 455) is not enabled, the compiler will suggest turning on the *DerivingStrategies* (page 455) extension.

-Wmissing-fields

Since at least 5.04

This option is on by default, and warns you whenever the construction of a labelled field constructor isn't complete, missing initialisers for one or more fields. While not an error (the missing fields are initialised with bottoms), it is often an indication of a programmer error.

-Wmissing-export-lists

Since 8.4

This flag warns if you declare a module without declaring an explicit export list. For example

```
module M where

  p x = x
```

The *-Wmissing-export-lists* (page 96) flag will warn that `M` does not declare an export list. Declaring an explicit export list for `M` enables GHC dead code analysis, prevents accidental export of names and can ease optimizations like inlining.

-Wmissing-import-lists

Since 7.0

This flag warns if you use an unqualified `import` declaration that does not explicitly list the entities brought into scope. For example

```
module M where
  import X( f )
  import Y
  import qualified Z
  p x = f x x
```

The *-Wmissing-import-lists* (page 96) flag will warn about the import of Y but not X If module Y is later changed to export (say) f, then the reference to f in M will become ambiguous. No warning is produced for the import of Z because extending Z's exports would be unlikely to produce ambiguity in M.

-Wmissing-methods

Since at least 5.04

Default on

This option warns you whenever an instance declaration is missing one or more methods, and the corresponding class declaration has no default declaration for them.

The MINIMAL pragma can be used to change which combination of methods will be required for instances of a particular class. See *MINIMAL pragma* (page 609).

-Wmissing-signatures

Since at least 5.04

Default off

If you would like GHC to check that every top-level function/value has a type signature, use the *-Wmissing-signatures* (page 97) option. As part of the warning GHC also reports the inferred type.

-Wmissing-exported-sigs

Since 7.10

This option is now deprecated in favour of *-Wmissing-exported-signatures* (page 97).

-Wmissing-exported-signatures

Since 8.0

Default off

If you would like GHC to check that every exported top-level function/value has a type signature, but not check unexported values, use the *-Wmissing-exported-signatures* (page 97) option. If this option is used in conjunction with *-Wmissing-signatures* (page 97) then every top-level function/value must have a type signature. As part of the warning GHC also reports the inferred type.

-Wmissing-local-sigs

Since 7.0

This option is now deprecated in favour of *-Wmissing-local-signatures* (page 97).

-Wmissing-local-signatures

Since 8.0

If you use the *-Wmissing-local-signatures* (page 97) flag GHC will warn you about any polymorphic local bindings. As part of the warning GHC also reports the inferred type. The option is off by default.

-Wmissing-pattern-synonym-signatures

Since 8.0

Default off

If you would like GHC to check that every pattern synonym has a type signature, use the *-Wmissing-pattern-synonym-signatures* (page 97) option. If this option is used in conjunction with *-Wmissing-exported-signatures* (page 97) then only exported pattern synonyms must have a type signature. GHC also reports the inferred type.

-Wmissing-kind-signatures

Since 9.2

Default off

If you would like GHC to check that every data, type family, type-class definition has a *standalone kind signature* (page 369) or a *CUSK* (page 367), use the *-Wmissing-kind-signatures* (page 98) option. You can specify the kind via *StandaloneKindSignatures* (page 369) or *CUSKs* (page 367).

Note that *-Wmissing-kind-signatures* (page 98) does not warn about associated type families, as GHC considers an associated type family declaration to have a CUSK if its enclosing class has a CUSK. (See *Complete user-supplied kind signatures and polymorphic recursion* (page 367) for more on this point.) Therefore, giving the parent class a standalone kind signature or CUSK is sufficient to fix the warning for the class's associated type families as well.

-Wmissing-poly-kind-signatures

Since 9.8

Default off

This is a restricted version of *-Wmissing-kind-signatures* (page 98).

It warns when a declaration defines a type constructor that lacks a *standalone kind signature* (page 369) and whose inferred kind is polymorphic (which happens with *-PolyKinds*. For example

```
data T a = MkT (a -> Int)    -- T :: Type -> Type
                             -- Not polymorphic, hence no warning
data W f a = MkW (f a)       -- W :: forall k. (k->Type) -> k -> Type
                             -- Polymorphic, hence warning!
```

It is useful to catch accidentally polykinded types, or to make that polymorphism explicit, without requiring a kind signature for every type.

-Wmissing-exported-pattern-synonym-signatures

Default off

If you would like GHC to check that every exported pattern synonym has a type signature, but not check unexported pattern synonyms, use the *-Wmissing-exported-pattern-synonym-signatures* (page 98) option. If this option is used in conjunction with *-Wmissing-pattern-synonym-signatures* (page 97) then every pattern synonym must have a type signature. As part of the warning GHC also reports the inferred type.

-Wname-shadowing

Since at least 5.04

This option causes a warning to be emitted whenever an inner-scope value has the same name as an outer-scope value, i.e. the inner value shadows the outer one. This can catch typographical errors that turn into hard-to-find bugs, e.g., in the inadvertent capture of what would be a recursive call in `f = ... let f = id in ... f ....`

The warning is suppressed for names beginning with an underscore. For example

```
f x = do { _ignore <- this; _ignore <- that; return (the other) }
```

-Worphans

Since 6.4

These flags cause a warning to be emitted whenever the module contains an "orphan" instance declaration or rewrite rule. An instance declaration is an orphan if it appears in a module in which neither the class nor the type being instanced are declared in the same module. A rule is an orphan if it is a rule for a function declared in another module. A module containing any orphans is called an orphan module.

The trouble with orphans is that GHC must pro-actively read the interface files for all orphan modules, just in case their instances or rules play a role, whether or not the module's interface would otherwise be of any use. See *Orphan modules and instance declarations* (page 218) for details.

The flag *-Worphans* (page 99) warns about user-written orphan rules or instances.

-Woverlapping-patterns

Since at least 5.04

By default, the compiler will warn you if a set of patterns are overlapping, e.g.,

```
f :: String -> Int
f []     = 0
f (_:xs) = 1
f "2"    = 2
```

where the last pattern match in `f` won't ever be reached, as the second pattern overlaps it. More often than not, redundant patterns is a programmer mistake/error, so this option is enabled by default.

If the programmer is dead set on keeping a redundant clause, for example to prevent bitrot, they can make use of a guard scrutinising `GHC.Exts.considerAccessible` to prevent the checker from flagging the parent clause as redundant:

```
g :: String -> Int
g []                       = 0
g (_:xs)                   = 1
g "2" | considerAccessible = 2 -- No warning!
```

Note that `considerAccessible` should come as the last statement of the guard in order not to impact the results of the checker. E.g., if you write

```
h :: Bool -> Int
h x = case (x, x) of
  (True,  True)  -> 1
  (False, False) -> 2
  (True,  False) | considerAccessible, False <- x -> 3
```

The pattern-match checker takes you by your word, will conclude that `False <- x` might fail and warn that the pattern-match is inexhaustive. Put `considerAccessible` last to avoid such confusions.

Note that due to technical limitations, `considerAccessible` will not suppress *-Winaccessible-code* (page 100) warnings.

-Winaccessible-code

Since 8.6

By default, the compiler will warn you if types make a branch inaccessible. This generally requires GADTs or similar extensions.

Take, for example, the following program

```
{-# LANGUAGE GADTs #-}

data Foo a where
 Foo1 :: Foo Char
 Foo2 :: Foo Int

data TyEquality a b where
        Refl :: TyEquality a a

checkTEQ :: Foo t -> Foo u -> Maybe (TyEquality t u)
checkTEQ x y = error "unimportant"

step2 :: Bool
step2 = case checkTEQ Foo1 Foo2 of
         Just Refl -> True -- Inaccessible code
         Nothing -> False
```

The `Just Refl` case in `step2` is inaccessible, because in order for `checkTEQ` to be able to produce a `Just`, `t ~ u` must hold, but since we're passing `Foo1` and `Foo2` here, it follows that `t ~ Char`, and `u ~ Int`, and thus `t ~ u` cannot hold.

-Wstar-is-type

Since 8.6

The use of * to denote the kind of inhabited types relies on the *StarIsType* (page 378) extension, which in a future release will be turned off by default and then possibly removed. The reasons for this and the deprecation schedule are described in GHC proposal #143.

This warning allows to detect such uses of * before the actual breaking change takes place. The recommended fix is to replace * with `Type` imported from `Data.Kind`.

-Wstar-binder

Since 8.6

Under *StarIsType* (page 378), a * in types is not an operator nor even a name, it is special syntax that stands for `Data.Kind.Type`. This means that an expression like `Either * Char` is parsed as `Either (*) Char` and not `(*) Either Char`.

In binding positions, we have similar parsing rules. Consider the following example

```
{-# LANGUAGE TypeOperators, TypeFamilies, StarIsType #-}

type family a + b
type family a * b
```

While `a + b` is parsed as `(+) a b` and becomes a binding position for the `(+)` type operator, `a * b` is parsed as `a (*) b` and is rejected.

As a workaround, we allow to bind `(*)` in prefix form:

```
type family (*) a b
```

This is a rather fragile arrangement, as generally a programmer expects `(*) a b` to be equivalent to `a * b`. With *-Wstar-binder* (page 100) we warn when this special treatment of `(*)` takes place.

-Wsimplifiable-class-constraints

Since 8.2

Default on

Warn about class constraints in a type signature that can be simplified using a top-level instance declaration. For example:

```
f :: Eq [a] => a -> a
```

Here the `Eq [a]` in the signature overlaps with the top-level instance for `Eq [a]`. GHC goes to some efforts to use the former, but if it should use the latter, it would then have an insoluble `Eq a` constraint. Best avoided by instead writing:

```
f :: Eq a => a -> a
```

-Wtabs

Since 6.8

Have the compiler warn if there are tabs in your source file.

-Wtype-defaults

Since at least 5.04

Default off

Have the compiler warn/inform you where in your source the Haskell defaulting mechanism for numeric types kicks in. This is useful information when converting code from a context that assumed one default into one with another, e.g., the 'default default' for Haskell 1.4 caused the otherwise unconstrained value `1` to be given the type `Int`, whereas Haskell 98 and later defaults it to `Integer`. This may lead to differences in performance and behaviour, hence the usefulness of being non-silent about this.

-Wmonomorphism-restriction

Since 6.8

Default off

Have the compiler warn/inform you where in your source the Haskell Monomorphism Restriction is applied. If applied silently the MR can give rise to unexpected behaviour, so it can be helpful to have an explicit warning that it is being applied.

-Wunsupported-llvm-version

Since 7.8

Warn when using *-fllvm* (page 243) with an unsupported version of LLVM.

-Wmissed-extra-shared-lib

Since 8.8

Warn when GHCi can't load a shared lib it deduced it should load when loading a package and analyzing the extra-libraries stanza of the target package description.

-Wunticked-promoted-constructors

Since 7.10

Warn if a promoted data constructor is used without a tick preceding its name.

For example:

```
data Nat = Succ Nat | Zero

data Vec n s where
  Nil  :: Vec Zero a
  Cons :: a -> Vec n a -> Vec (Succ n) a
```

Will raise two warnings because `Zero` and `Succ` are not written as `'Zero` and `'Succ`.

This also applies to list literals since 9.4. For example:

```
type L = [Int, Char, Bool]
```

will raise a warning, because `[Int, Char, Bool]` is a promoted list which lacks a tick.

-Wunused-binds

Since at least 5.04

Report any function definitions (and local bindings) which are unused. An alias for

- *-Wunused-top-binds* (page 102)
- *-Wunused-local-binds* (page 102)
- *-Wunused-pattern-binds* (page 103)

-Wunused-top-binds

Since 8.0

Report any function definitions which are unused.

More precisely, warn if a binding brings into scope a variable that is not used, except if the variable's name starts with an underscore. The "starts-with-underscore" condition provides a way to selectively disable the warning.

A variable is regarded as "used" if

- It is exported, or
- It appears in the right hand side of a binding that binds at least one used variable that is used

For example:

```
module A (f) where
f = let (p,q) = rhs1 in t p  -- No warning: q is unused, but is locally bound
t = rhs3                     -- No warning: f is used, and hence so is t
g = h x                      -- Warning: g unused
h = rhs2                     -- Warning: h is only used in the
                             -- right-hand side of another unused binding
_w = True                    -- No warning: _w starts with an underscore
```

-Wunused-local-binds

Since 8.0

Report any local definitions which are unused. For example:

```
module A (f) where
f = let (p,q) = rhs1 in t p  -- Warning: q is unused
g = h x                      -- No warning: g is unused, but is a top-level
↪binding
```

-Wunused-pattern-binds

Since 8.0

Warn if a pattern binding binds no variables at all, unless it is a lone wild-card pattern, or a banged pattern. For example:

```
Just _ = rhs3    -- Warning: unused pattern binding
(_, _) = rhs4    -- Warning: unused pattern binding
_  = rhs3        -- No warning: lone wild-card pattern
!() = rhs4       -- No warning: banged pattern; behaves like seq
```

In general a lazy pattern binding *p* = *e* is a no-op if *p* does not bind any variables. The motivation for allowing lone wild-card patterns is they are not very different from `_v = rhs3`, which elicits no warning; and they can be useful to add a type constraint, e.g. `_ = x::Int`. A banged pattern (see *Bang patterns and Strict Haskell* (page 540)) is *not* a no-op, because it forces evaluation, and is useful as an alternative to `seq`.

-Wunused-imports

Since at least 5.04

Report any modules that are explicitly imported but never used. However, the form `import M()` is never reported as an unused import, because it is a useful idiom for importing instance declarations, which are anonymous in Haskell.

-Wunused-matches

Since at least 5.04

Report all unused variables which arise from term-level pattern matches, including patterns consisting of a single variable. For instance `f x y = []` would report `x` and `y` as unused. The warning is suppressed if the variable name begins with an underscore, thus:

```
f _x = True
```

Note that *-Wunused-matches* (page 103) does not warn about variables which arise from type-level patterns, as found in type family and data family instances. This must be enabled separately through the *-Wunused-type-patterns* (page 104) flag.

-Wunused-do-bind

Since 6.12

Report expressions occurring in `do` and `mdo` blocks that appear to silently throw information away. For instance `do { mapM popInt xs ; return 10 }` would report the first statement in the `do` block as suspicious, as it has the type `StackM [Int]` and not `StackM ()`, but that `[Int]` value is not bound to anything. The warning is suppressed by explicitly mentioning in the source code that your program is throwing something away:

```
do { _ <- mapM popInt xs ; return 10 }
```

Of course, in this particular situation you can do even better:

```
do { mapM_ popInt xs ; return 10 }
```

-Wunused-type-patterns

Since 8.0

Report all unused implicitly bound type variables which arise from patterns in type family and data family instances. For instance:

```
type instance F x y = []
```

would report `x` and `y` as unused on the right hand side. The warning is suppressed if the type variable name begins with an underscore, like so:

```
type instance F _x _y = []
```

When *ExplicitForAll* (page 506) is enabled, explicitly quantified type variables may also be identified as unused. For instance:

```
type instance forall x y. F x y = []
```

would still report `x` and `y` as unused on the right hand side

Unlike *-Wunused-matches* (page 103), *-Wunused-type-patterns* (page 104) is not implied by *-Wall* (page 86). The rationale for this decision is that unlike term-level pattern names, type names are often chosen expressly for documentation purposes, so using underscores in type names can make the documentation harder to read.

-Wunused-foralls

Since 8.0

Report all unused type variables which arise from explicit, user-written `forall` statements. For instance:

```
g :: forall a b c. (b -> b)
```

would report `a` and `c` as unused.

-Wunused-record-wildcards

Since 8.10

Report all record wildcards where none of the variables bound implicitly are used. For instance:

```
data P = P { x :: Int, y :: Int }

f1 :: P -> Int
f1 P{..} = 1 + 3
```

would report that the `P{..}` match is unused.

-Wredundant-bang-patterns

Since 9.2

Report dead bang patterns, where dead bangs are bang patterns that under no circumstances can force a thunk that wasn't already forced. Dead bangs are a form of redundant bangs. The new check is performed in pattern-match coverage checker along with other checks (namely, redundant and inaccessible RHSs). Given

```
f :: Bool -> Int
f True = 1
f !x   = 2
```

The bang pattern on `!x` is dead. By the time the `x` in the second equation is reached, `x` will already have been forced due to the first equation (`f True = 1`). Moreover, there is no way to reach the second equation without going through the first one.

Note that `-Wredundant-bang-patterns` will not warn about dead bangs that appear on a redundant clause. That is because in that case, it is recommended to delete the clause wholly, including its leading pattern match.

Dead bang patterns are redundant. But there are bang patterns which are redundant that aren't dead, for example:

```
f !() = 0
```

the bang still forces the argument, before we attempt to match on `()`. But it is redundant with the forcing done by the `()` match. Currently such redundant bangs are not considered dead, and `-Wredundant-bang-patterns` will not warn about them.

-Wredundant-record-wildcards

Since 8.10

Report all record wildcards where the wild card match binds no patterns. For instance:

```
data P = P { x :: Int, y :: Int }

f1 :: P -> Int
f1 P{x,y,..} = x + y
```

would report that the `P{x, y, ..}` match has a redundant use of `..`.

-Wredundant-strictness-flags

Since 9.4

Report strictness flags applied to unlifted types. An unlifted type is always strict, and applying a strictness flag has no effect.

For example:

```
data T = T !Int#
```

-Wwrong-do-bind

Since 6.12

Report expressions occurring in `do` and `mdo` blocks that appear to lack a binding. For instance `do { return (popInt 10) ; return 10 }` would report the first statement in the `do` block as suspicious, as it has the type `StackM (StackM Int)` (which consists of two nested applications of the same monad constructor), but which is not then "unpacked" by binding the result. The warning is suppressed by explicitly mentioning in the source code that your program is throwing something away:

```
do { _ <- return (popInt 10) ; return 10 }
```

For almost all sensible programs this will indicate a bug, and you probably intended to write:

```
do { popInt 10 ; return 10 }
```

-Winline-rule-shadowing

Since 7.8

Warn if a rewrite RULE might fail to fire because the function might be inlined before the rule has a chance to fire. See *How rules interact with INLINE/NOINLINE pragmas* (page 593).

-Wcpp-undef

Since 8.2

This flag passes `-Wundef` to the C pre-processor (if its being used) which causes the pre-processor to warn on uses of the *#if* directive on undefined identifiers.

-Wunbanged-strict-patterns

Since 8.2

This flag warns whenever you write a pattern that binds a variable whose type is unlifted, and yet the pattern is not a bang pattern nor a bare variable. See *Unboxed types* (page 554) for information about unlifted types.

-Wmissing-home-modules

Since 8.2

When a module provided by the package currently being compiled (i.e. the "home" package) is imported, but not explicitly listed in command line as a target. Useful for Cabal to ensure GHC won't pick up modules, not listed neither in `exposed-modules`, nor in `other-modules`.

-Wpartial-fields

Since 8.4

The option *-Wpartial-fields* (page 106) warns about a record field `f` that is defined in some, but not all, of the constructors of a data type, as such selector functions are partial. For example, when *-Wpartial-fields* (page 106) is enabled the compiler will emit a warning at the definition of `Foo` below:

```
data Foo = Foo { f :: Int } | Bar
```

The warning is suppressed if the field name begins with an underscore.

```
data Foo = Foo { _f :: Int } | Bar
```

Another related warning is *-Wincomplete-record-selectors* (page 95), which warns at use sites rather than definition sites.

-Wunused-packages

Since 8.10

The option *-Wunused-packages* (page 106) warns about packages, specified on command line via *-package ⟨pkg⟩* (page 221) or *-package-id ⟨unit-id⟩* (page 221), but were not needed during compilation. If the warning fires it means the specified package wasn't needed for compilation.

This warning interacts poorly with GHCi because most invocations will pass a large number of `-package` arguments on the initial load. Therefore if you modify the targets using `:load` or `:cd` then the warning will be silently disabled if it's enabled (see #21110).

-Winvalid-haddock

Since 9.0

When the `-haddock` option is enabled, GHC collects documentation comments and associates them with declarations, function arguments, data constructors, and other syntactic elements. Documentation comments in invalid positions are discarded:

```
myValue =
  -- | Invalid (discarded) comment in an expression
  2 + 2
```

This warning informs you about discarded documentation comments. It has no effect when *-haddock* (page 84) is disabled.

-Woperator-whitespace-ext-conflict

Since 9.2

When *TemplateHaskell* (page 528) is enabled, `f $x` is parsed as `f` applied to an untyped splice. But when the extension is disabled, the expression is parsed as a use of the `$` infix operator.

To make it easy to read `f $x` without checking the enabled extensions, one could rewrite it as `f $ x`, which is what this warning suggests.

Currently, it detects the following cases:

- `$x` could mean an untyped splice under *TemplateHaskell* (page 528)
- `$$x` could mean a typed splice under *TemplateHaskell* (page 528)
- `%m` could mean a multiplicity annotation under *LinearTypes* (page 408)

It only covers extensions that currently exist. If you want to enforce a stricter policy and always require whitespace around all infix operators, use *-Woperator-whitespace* (page 107).

-Woperator-whitespace

Since 9.2

There are four types of infix operator occurrences, as defined by GHC Proposal #229:

```
a ! b   -- a loose infix occurrence
a!b     -- a tight infix occurrence
a !b    -- a prefix occurrence
a! b    -- a suffix occurrence
```

A loose infix occurrence of any operator is always parsed as an infix operator, but other occurrence types may be assigned a special meaning. For example, a prefix `!` denotes a bang pattern, and a prefix `$` denotes a *TemplateHaskell* (page 528) splice.

This warning encourages the use of loose infix occurrences of all infix operators, to prevent possible conflicts with future language extensions.

-Wauto-orphans

Since 7.4

Does nothing.

-Wmissing-space-after-bang

Since 8.8

Does nothing.

-Wderiving-typeable

Since 7.10

This flag warns when `Typeable` is listed in a deriving clause or derived with *StandaloneDeriving* (page 436).

Since GHC 7.10, `Typeable` is automatically derived for all types. Thus, deriving `Typeable` yourself is redundant.

-Wambiguous-fields

Since 9.2

When *DuplicateRecordFields* (page 425) is enabled, the option *-Wambiguous-fields* (page 108) warns about occurrences of fields in selectors or updates that depend on the deprecated mechanism for type-directed disambiguation. This mechanism will be removed in a future GHC release, at which point these occurrences will be rejected as ambiguous. See the proposal DuplicateRecordFields without ambiguous field access and the documentation on *DuplicateRecordFields* (page 425) for further details.

This warning has no effect when *DuplicateRecordFields* (page 425) is disabled.

-Wforall-identifier

Since 9.4

This warning is deprecated. It no longer has any effect since GHC 9.10.

In the past, GHC used to accept `forall` as a term-level identifier:

```
-- from constraints-0.13
forall :: forall p. (forall a. Dict (p a)) -> Dict (Forall p)
forall d = ...
```

In accordance with GHC Proposal #281, this is no longer possible, as `forall` has become a proper keyword. *-Wforall-identifier* (page 108) was used in the migration period before the breaking change took place.

-Wunicode-bidirectional-format-characters

Since 9.0.2

Explicit unicode bidirectional formatting characters can cause source code to be rendered misleadingly in many viewers. We warn if any such character is present in the source.

Specifically, the characters disallowed by this warning are those which are a part of the 'Explicit Formatting` category of the Unicode Bidirectional Character Type Listing

-Wgadt-mono-local-binds

Since 9.4.1

This warning is triggered on pattern matching involving GADTs, if *MonoLocalBinds* (page 526) is disabled. Type inference can be fragile in this case.

See the OutsideIn(X) paper (section 4.2) and *Let-generalisation* (page 526) for more details.

To resolve this warning, you can enable *MonoLocalBinds* (page 526) or an extension implying it (*GADTs* (page 335) or *TypeFamilies* (page 338)).

The warning is also triggered when matching on GADT-like pattern synonyms (i.e. pattern synonyms containing equalities in provided constraints).

In previous versions of GHC (9.2 and below), it was an error to pattern match on a GADT if neither *GADTs* (page 335) nor *TypeFamilies* (page 338) were enabled.

-Wtype-equality-out-of-scope

Since 9.4.1

Default on

In accordance with GHC Proposal #371, the type equality syntax `a ~ b` is no longer built-in. Instead, ~ is a regular type operator that can be imported from `Data.Type.Equality` or `Prelude`.

To minimize breakage, a compatibility fallback is provided: whenever ~ is used but is not in scope, the compiler assumes that it stands for a type equality constraint. The warning is triggered by any code that relies on this fallback. It can be addressed by bringing ~ into scope explicitly.

The likely culprit is that you use *NoImplicitPrelude* (page 295) and a custom Prelude. In this case, consider updating your custom Prelude to re-export ~ from `Data.Type.Equality`.

-Wtype-equality-requires-operators

Since 9.4.1

In accordance with GHC Proposal #371, the type equality syntax `a ~ b` is no longer built-in. Instead, ~ is a regular type operator that requires the *TypeOperators* (page 323) extension.

To minimize breakage, ~ specifically (unlike other type operators) can be used even when *TypeOperators* (page 323) is disabled. The warning is triggered whenever this happens, and can be addressed by enabling the extension.

-Wloopy-superclass-solve

Since 9.6.1

This warning is deprecated. It no longer has any effect since GHC 9.10. In the past, *UndecidableInstances* (page 484) allowed potentially non-terminating evidence for certain superclass constraints. This is no longer allowed, as explained in *Undecidable instances and loopy superclasses* (page 484). This warning was used during the transition period.

-Wterm-variable-capture

Since 9.8.1

Under *RequiredTypeArguments* (page 395), implicit quantification of type variables does not take place if there is a term variable of the same name in scope.

For example:

```
a = 15
f :: a -> a    -- NoRequiredTypeArguments: The ‘a’ is implicitly quantified
               -- RequiredTypeArguments:   The ‘a’ refers to the term-level␣
↪binding
```

When *-Wterm-variable-capture* (page 109) is enabled, GHC warns against implicit quantification that would stop working under *RequiredTypeArguments* (page 395).

-Wmissing-role-annotations

Since 9.8.1

Default off

If you would like GHC to check that every data type definition has a *role annotation* (page 419), use the *-Wmissing-role-annotations* (page 110) option. You can specify the role via *RoleAnnotations* (page 419).

GHC will not warn about type class definitions with missing role annotations, as their default roles are the strictest: all nominal. In other words the type-class role cannot be accidentally left representational or phantom, which could affected the code correctness.

-Wimplicit-rhs-quantification

Since 9.8.1

Default off

In accordance with GHC Proposal #425, GHC will stop implicitly quantifying over type variables that occur free on the right-hand side of a type synonym but are not mentioned on the left-hand side. Type synonym declarations that rely on this form of quantification should be rewritten with invisible binders.

For example:

```
type T1 :: forall a . Maybe a
type T1    = 'Nothing :: Maybe a      -- old
type T1 @a = 'Nothing :: Maybe a      -- new
```

This warning detects code that will be affected by this breaking change.

-Wdeprecated-type-abstractions

Since 9.10.1

Default off

Type abstractions in constructor patterns allow binding existential type variables:

```
import Type.Reflection (Typeable, typeRep)
data Ex = forall e. (Typeable e, Show e) => MkEx e
showEx (MkEx @e a) = show a ++ " :: " ++ show (typeRep @e)
```

Note the pattern `MkEx @e a`, and specifically the `@e` binder.

Support for this feature was added to GHC in version 9.2, but instead of getting its own language extension the feature was enabled by a combination of *TypeApplications* (page 386) and *ScopedTypeVariables* (page 512). As per GHC Proposal #448 and its amendment #604 we are now transitioning towards guarding this feature behind *TypeAbstractions* (page 390) instead.

As a compatibility measure, GHC continues to support old programs that use type abstractions in constructor patterns without enabling the appropriate extension *TypeAbstractions* (page 390), but it will stop doing so in a future release.

This warning detects code that will be affected by this breaking change.

-Wincomplete-export-warnings

Since 9.8.1

Ino accordance with GHC Proposal #134, it is now possible to deprecate certain exports of a name without deprecating the name itself.

As explained in *WARNING and DEPRECATED pragmas* (page 606), when a name is exported in several ways in the same module, but only some of those ways have a warning, it will not end up deprecated when imported in another module.

For example:

```
   module A (x) where

   x :: Int
   x = 2

   module M (
       {-# WARNING x "deprecated" #-} x
       module A
     )
   import A

When :ghc-flag:`-Wincomplete-export-warnings` is enabled, GHC warns about exports
that are not deprecating a name that is deprecated with another export in that␣
↪module.
```

-Wbadly-staged-types

Since 9.10.1

Consider an example:

```
tardy :: forall a. Proxy a -> IO Type
tardy _ = [t| a |]
```

The type binding `a` is bound at stage 1 but used on stage 2.

This is badly staged program, and the `tardy (Proxy @Int)` won't produce a type representation of `Int`, but rather a local name `a`.

-Winconsistent-flags

Since 9.8.1

Default on

Warn when command line options are inconsistent in some way.

For example, when using GHCi, optimisation flags are ignored and a warning is issued. Another example is *-dynamic* (page 246) is ignored when *-dynamic-too* (page 245) is passed.

-Wdata-kinds-tc

Since 9.10.1

Introduced in GHC 9.10.1, this warns when an illegal use of a type or kind (without having enabled the *DataKinds* (page 357) extension) is caught in the typechecker (hence the `-tc` suffix). These warnings complement the existing *DataKinds* (page 357) checks (that have existed since *DataKinds* (page 357) was first introduced), which result in errors instead of warnings.

This warning is scheduled to be changed to an error in a future GHC version, at which point the *-Wdata-kinds-tc* (page 111) flag will be removed. Users can enable the *DataKinds* (page 357) extension to avoid issues (thus silencing the warning).

-Wdefaulted-exception-context

Since 9.10.1

Introduced in GHC 9.10.1 with the introduction of an implicit Control.Exception.Context.ExceptionContext` context to Control.Exception.SomeException. To preserve compatibility with earlier compilers, this constraints is implicitly defaulted to Control.Exception.Context.emptyExceptionContext when no other evidence is available. As this behavior may result in dropped exception context this warning is provided to give notice when defaulting occurs.

If you're feeling really paranoid, the *-dcore-lint* (page 265) option is a good choice. It turns on heavyweight intra-pass sanity-checking within GHC. (It checks GHC's sanity, not yours.)

5.3 Optimisation (code improvement)

The `-O*` options specify convenient "packages" of optimisation flags; the `-f*` options described later on specify *individual* optimisations to be turned on/off; the `-m*` options specify *machine-specific* optimisations to be turned on/off.

Most of these options are boolean and have options to turn them both "on" and "off" (beginning with the prefix `no-`). For instance, while `-fspecialise` enables specialisation, `-fno-specialise` disables it. When multiple flags for the same option appear in the command-line they are evaluated from left to right. For instance, `-fno-specialise -fspecialise` will enable specialisation.

It is important to note that the `-O*` flags are roughly equivalent to combinations of `-f*` flags. For this reason, the effect of the `-O*` and `-f*` flags is dependent upon the order in which they occur on the command line.

For instance, take the example of `-fno-specialise -O1`. Despite the `-fno-specialise` appearing in the command line, specialisation will still be enabled. This is the case as `-O1` implies `-fspecialise`, overriding the previous flag. By contrast, `-O1 -fno-specialise` will compile without specialisation, as one would expect.

5.3.1 -O*: convenient "packages" of optimisation flags.

There are *many* options that affect the quality of code produced by GHC. Most people only have a general goal, something like "Compile quickly" or "Make my program run like greased lightning." The following "packages" of optimisations (or lack thereof) should suffice.

Note that higher optimisation levels cause more cross-module optimisation to be performed, which can have an impact on how much of your program needs to be recompiled when you change something. This is one reason to stick to no-optimisation when developing code.

No ``-O*``-type option specified: This is taken to mean "Please compile quickly; I'm not over-bothered about compiled-code quality." So, for example, `ghc -c Foo.hs`

`-O0`
Means "turn off all optimisation", reverting to the same settings as if no `-O` options had been specified. Saying `-O0` can be useful if e.g. `make` has inserted a `-O` on the command line already.

`-O`
`-O1`
Means: "Generate good-quality code without taking too long about it." Thus, for example: `ghc -c -O Main.lhs`

`-O2`
Means: "Apply every non-dangerous optimisation, even if it means significantly longer compile times."

The avoided "dangerous" optimisations are those that can make runtime or space *worse* if you're unlucky. They are normally turned on or off individually.

`-O⟨n⟩`
Any -On where n > 2 is the same as -O2.

We don't use a `-O*` flag for day-to-day work. We use `-O` to get respectable speed; e.g., when we want to measure something. When we want to go for broke, we tend to use `-O2` (and we go for lots of coffee breaks).

The easiest way to see what `-O` (etc.) "really mean" is to run with *`-v`* (page 76), then stand back in amazement.

5.3.2 -f*: platform-independent flags

These flags turn on and off individual optimisations. Flags marked as *on* by default are enabled at all optimisation levels by default, and as such you shouldn't need to set any of them explicitly. A flag `-fwombat` can be negated by saying `-fno-wombat`.

`-fcore-constant-folding`

Default off but enabled by *`-O`* (page 113).

Enables Core-level constant folding, i.e. propagation of values that can be computed at compile time.

`-fcase-merge`

Default off but enabled by *`-O`* (page 113).

Merge immediately-nested case expressions that scrutinise the same variable. For example,

```
case x of
   Red -> e1
   _   -> case x of
            Blue -> e2
            Green -> e3
```

Is transformed to,

```
case x of
   Red -> e1
   Blue -> e2
   Green -> e2
```

-fcase-folding

Default off but enabled by *-O* (page 113).

Allow constant folding in case expressions that scrutinise some primops: For example,

```
case x `minusWord#` 10## of
   10## -> e1
   20## -> e2
   v    -> e3
```

Is transformed to,

```
case x of
   20## -> e1
   30## -> e2
   _    -> let v = x `minusWord#` 10## in e3
```

-fcall-arity

Default off but enabled by *-O* (page 113).

Enable call-arity analysis.

-fexitification

Default off but enabled by *-O* (page 113).

Enables the floating of exit paths out of recursive functions.

-fcmm-elim-common-blocks

Default off but enabled by *-O* (page 113).

Enables the common block elimination optimisation in the code generator. This optimisation attempts to find identical Cmm blocks and eliminate the duplicates.

-fcmm-sink

Default off but enabled by *-O* (page 113).

Enables the sinking pass in the code generator. This optimisation attempts to find identical Cmm blocks and eliminate the duplicates attempts to move variable bindings closer to their usage sites. It also inlines simple expressions like literals or registers.

-fcmm-static-pred

Default off but enabled by *-O* (page 113).

This enables static control flow prediction on the final Cmm code. If enabled GHC will apply certain heuristics to identify loops and hot code paths. This information is then used by the register allocation and code layout passes.

-fcmm-control-flow

Default off but enabled by *-O* (page 113).

Enables some control flow optimisations in the Cmm code generator, merging basic blocks and avoiding jumps right after jumps.

-fasm-shortcutting

Default off but enabled by *-O2* (page 113).

This enables shortcutting at the assembly stage of the code generator. In simpler terms shortcutting means if a block of instructions A only consists of a unconditionally jump, we replace all jumps to A by jumps to the successor of A.

This is mostly done during Cmm passes. However this can miss corner cases. So at `-O2` this flag runs the pass again at the assembly stage to catch these. Note that due to platform limitations (#21972) this flag does nothing on macOS.

-fblock-layout-cfg

Default off but enabled by *-O* (page 113).

The new algorithm considers all outgoing edges of a basic blocks for code layout instead of only the last jump instruction. It also builds a control flow graph for functions, tries to find hot code paths and place them sequentially leading to better cache utilization and performance.

This is expected to improve performance on average, but actual performance difference can vary.

If you find cases of significant performance regressions, which can be traced back to obviously bad code layout please open a ticket.

-fblock-layout-weights

This flag is hacker territory. The main purpose of this flag is to make it easy to debug and tune the new code layout algorithm. There is no guarantee that values giving better results now won't be worse with the next release.

If you feel your code warrants modifying these settings please consult the source code for default values and documentation. But I strongly advise against this.

-fblock-layout-weightless

Default off

When not using the cfg based blocklayout layout is determined either by the last jump in a basic block or the heaviest outgoing edge of the block in the cfg.

With this flag enabled we use the last jump instruction in blocks. Without this flags the old algorithm also uses the heaviest outgoing edge.

When this flag is enabled and *-fblock-layout-cfg* (page 115) is disabled block layout behaves the same as in 8.6 and earlier.

-fcpr-anal

Default off but enabled by *-O* (page 113).

Turn on CPR analysis, which enables the worker/wrapper transformation (cf. *-fworker-wrapper* (page 130)) to unbox the result of a function, such as

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

CPR analysis will see that each code path produces a *constructed product* such as `I# 0#` in the first branch (where `GHC.Exts.I#` is the data constructor of `Int`, boxing up the primitive integer literal `0#` of type `Int#`) and optimise to

```
sum xs = I# ($wsum xs)
$wsum []        = 0#
$wsum (I# x:xs) = x# +# $wsum xs
```

and then `sum` can inline to potentially cancel away the `I#` box.

Here's an example of the function that *does not* return a constructed product:

```
f :: [Int] -> (Int -> Int) -> Int
f []     g = g 0
f (x:xs) g = x + f xs g
```

The expression `g 0` is not a constructed product, because we don't know anything about `g`.

CPR analysis also works *nestedly*, for example

```
sumIO :: [Int] -> IO Int
sumIO []     = return 0
sumIO (x:xs) = do
  r <- sumIO xs
  return $! x + r
```

Note the use of `$!`: Without it, GHC would be unable to see that evaluation of `r` and `x` terminates (and rapidly, at that). An alternative would be to evaluate both with a bang pattern or a `seq`, but the `return $! <res>` idiom should work more reliably and needs less thinking. The above example will be optimised to

```
sumIO :: [Int] -> IO Int
sumIO xs = IO $ \s -> case $wsum xs s of
  (# s', r #) -> (# s', I# r #)
$wsumIO :: [Int] -> (# State# RealWorld, Int# #)
$wsumIO []        s = (# s, 0# #)
$wsumIO (I# x:xs) s = case $wsumIO xs s of
  (# s', r #) -> (# s', x +# r#)
```

And the latter can inline `sumIO` and cancel away the `I#` constructor. Unboxing the result of a `State` action should work similarly.

-fcse

Default off but enabled by *-O* (page 113).

Enables the common-sub-expression elimination optimisation. Switching this off can be useful if you have some `unsafePerformIO` expressions that you don't want commoned-up.

-fstg-cse

Default off but enabled by *-O* (page 113).

Enables the common-sub-expression elimination optimisation on the STG intermediate language, where it is able to common up some subexpressions that differ in their types, but not their representation.

-fdicts-cheap

Default off

A very experimental flag that makes dictionary-valued expressions seem cheap to the optimiser.

-fdicts-strict

Default off but enabled by *-O2* (page 113).

Make dictionaries strict.

This enables WW to fire on dictionary constraints which usually results in better runtime. In niche cases it can lead to significant compile time regressions because of changed inlining behaviour. Rarely this can also affect runtime negatively.

If enabling this flag leads to regressions try increasing the unfolding threshold using *-funfolding-use-threshold=⟨n⟩* (page 129) by a modest amount (~30) as this is likely a result of a known limitation described in *#18421*.

-fdmd-tx-dict-sel

Default on

Use a special demand transformer for dictionary selectors. Behaviour is unconditionally enabled starting with 9.2

-fdo-eta-reduction

Default on

Eta-reduce lambda expressions, if doing so gets rid of a whole group of lambdas.

-fdo-lambda-eta-expansion

Default on

Eta-expand let-bindings to increase their arity.

-fdo-clever-arg-eta-expansion

Default off

Eta-expand arguments to increase their arity to avoid allocating unnecessary thunks for them.

-feager-blackholing

Default off

Usually GHC black-holes a thunk only when it switches threads. This flag makes it do so as soon as the thunk is entered. See Haskell on a shared-memory multiprocessor.

See *Compile-time options for SMP parallelism* (page 132) for a discussion on its use.

-fexcess-precision

Default off

When this option is given, intermediate floating point values can have a *greater* precision/range than the final type. Generally this is a good thing, but some programs may rely on the exact precision/range of `Float`/`Double` values and should not use this option for their compilation.

Note that the 32-bit x86 native code generator only supports excess-precision mode, so neither `-fexcess-precision` nor `-fno-excess-precision` has any effect. This is a known bug, see *Bugs in GHC* (page 739).

-fexpose-all-unfoldings

Default off

An experimental flag to expose all unfoldings, even for very large or recursive functions. This allows for all functions to be inlined while usually GHC would avoid inlining larger functions.

-ffloat-in

Default off but enabled by *-O* (page 113).

Float let-bindings inwards, nearer their binding site. See Let-floating: moving bindings to give faster programs (ICFP'96).

This optimisation moves let bindings closer to their use site. The benefit here is that this may avoid unnecessary allocation if the branch the let is now on is never executed. It also enables other optimisation passes to work more effectively as they have more information locally.

This optimisation isn't always beneficial though (so GHC applies some heuristics to decide when to apply it). The details get complicated but a simple example is that it is often beneficial to move let bindings outwards so that multiple let bindings can be grouped into a larger single let binding, effectively batching their allocation and helping the garbage collector and allocator.

-ffull-laziness

Default off but enabled by *-O* (page 113).

Run the full laziness optimisation (also known as let-floating), which floats let-bindings outside enclosing lambdas, in the hope they will be thereby be computed less often. See Let-floating: moving bindings to give faster programs (ICFP'96). Full laziness increases sharing, which can lead to increased memory residency.

Note: GHC doesn't implement complete full laziness. Although GHC's full-laziness optimisation does enable some transformations which would be performed by a fully lazy implementation (such as extracting repeated computations from loops), these transformations are not applied consistently, so don't rely on them.

-ffun-to-thunk

Default off

Worker/wrapper removes unused arguments, but usually we do not remove them all, lest it turn a function closure into a thunk, thereby perhaps creating a space leak and/or disrupting inlining. This flag allows worker/wrapper to remove *all* value lambdas.

This flag was ineffective in the presence of *-ffull-laziness* (page 118), which would flout a thunk out of a constant worker function *even though* *-ffun-to-thunk* (page 118) was off.

Hence use of this flag is deprecated since GHC 9.4.1 and we rather suggest to pass `-fno-full-laziness` instead. That implies there's no way for worker/wrapper to turn a function into a thunk in the presence of `-fno-full-laziness`. If that is inconvenient for you, please leave a comment on the issue tracker (#21204).

-fignore-asserts

Default off but enabled by *-O* (page 113).

Causes GHC to ignore uses of the function `Exception.assert` in source code (in other words, rewriting `Exception.assert p e` to `e` (see *Assertions* (page 601)).

-fignore-interface-pragmas

Default Implied by *-O0* (page 113), otherwise off.

Tells GHC to ignore all inessential information when reading interface files. That is, even if `M.hi` contains unfolding or strictness information for a function, GHC will ignore that information.

-fkeep-auto-rules

Default off

The type-class specialiser and call-pattern specialisation both generate so-called "auto" RULES. These rules are usually exposed to importing modules in the interface file. But an auto rule is the sole reason for keeping a function alive, both the rule and the function are discarded, by default. That reduces code bloat, but risks the same function being specialised again in an importing module.

You can change this behaviour with *-fkeep-auto-rules* (page 119). Switching it on keeps all auto-generated rules.

-flate-dmd-anal

Default off

Run demand analysis again, at the end of the simplification pipeline. We found some opportunities for discovering strictness that were not visible earlier; and optimisations like *-fspec-constr* (page 122) can create functions with unused arguments which are eliminated by late demand analysis. Improvements are modest, but so is the cost. See notes on the wiki page.

-fliberate-case

Default off but enabled by *-O2* (page 113).

Turn on the liberate-case transformation. This unrolls recursive function once in its own RHS, to avoid repeated case analysis of free variables. It's a bit like the call-pattern specialiser (*-fspec-constr* (page 122)) but for free variables rather than arguments.

-fliberate-case-threshold=⟨n⟩

Default 2000

Set the size threshold for the liberate-case transformation.

-floopification

Default off but enabled by *-O* (page 113).

When this optimisation is enabled the code generator will turn all self-recursive saturated tail calls into local jumps rather than function calls.

-fllvm-pass-vectors-in-regs

Default on

This flag has no effect since GHC 8.8 - its behavior is always on. It used to instruct GHC to use the platform's native vector registers to pass vector arguments during function calls.

-fmax-inline-alloc-size=⟨n⟩

Default 128

Set the maximum size of inline array allocations to n bytes. GHC will allocate non-pinned arrays of statically known size in the current nursery block if they're no bigger than n bytes, ignoring GC overheap. This value should be quite a bit smaller than the block size (typically: 4096).

-fmax-inline-memcpy-insns=⟨n⟩

Default 32

Inline `memcpy` calls if they would generate no more than ⟨n⟩ pseudo-instructions.

-fmax-inline-memset-insns=⟨n⟩

Default 32

Inline `memset` calls if they would generate no more than n pseudo instructions.

-fmax-relevant-binds=⟨n⟩

Default 6

The type checker sometimes displays a fragment of the type environment in error messages, but only up to some maximum number, set by this flag. Turning it off with `-fno-max-relevant-binds` gives an unlimited number. Syntactically top-level bindings are also usually excluded (since they may be numerous), but `-fno-max-relevant-binds` includes them too.

-fmax-uncovered-patterns=⟨n⟩

Default 4

Maximum number of unmatched patterns to be shown in warnings generated by *-Wincomplete-patterns* (page 94) and *-Wincomplete-uni-patterns* (page 94).

-fmax-simplifier-iterations=⟨n⟩

Default 4

Sets the maximal number of iterations for the simplifier.

-flocal-float-out

Default on

Enable local floating of bindings from the RHS of a let(rec) in the simplifier. For example

```
let x = let y = rhs_y in rhs_x in blah
==>
let y = rhs_y in let x = rhs_x in blah
```

See the paper "Let-floating: moving bindings to give faster programs", Partain, Santos, and Peyton Jones; ICFP 1996. https://www.microsoft.com/en-us/research/publication/let-floating-moving-bindings-to-give-faster-programs/

Note: This is distinct from the global floating pass which can be disabled with *-fno-full-laziness* (page 118).

-flocal-float-out-top-level

Default on

Enable local floating of top-level bindings from the RHS of a let(rec) in the simplifier. For example

x = let y = e in (a,b) ===> y = e; x = (a,b)

See the paper "Let-floating: moving bindings to give faster programs", Partain, Santos, and Peyton Jones; ICFP 1996. https://www.microsoft.com/en-us/research/publication/let-floating-moving-bindings-to-give-faster-programs/

Note that if *-fno-local-float-out* (page 120) is set, that will take precedence.

Note: This is distinct from the global floating pass which can be disabled with *-fno-full-laziness* (page 118).

-fmax-worker-args=⟨n⟩

Default 10

A function will not be split into worker and wrapper if the number of value arguments of the resulting worker exceeds both that of the original function and this setting.

-fno-opt-coercion

Default coercion optimisation enabled.

Turn off the coercion optimiser.

-fno-pre-inlining

Default pre-inlining enabled

Turn off pre-inlining.

-fno-state-hack

Default state hack is enabled

Turn off the "state hack" whereby any lambda with a `State#` token as argument is considered to be single-entry, hence it is considered okay to inline things inside it. This can improve performance of IO and ST monad code, but it runs the risk of reducing sharing.

-fomit-interface-pragmas

Default Implied by *-O0* (page 113), otherwise off.

Tells GHC to omit all inessential information from the interface file generated for the module being compiled (say M). This means that a module importing M will see only the *types* of the functions that M exports, but not their unfoldings, strictness info, etc. Hence, for example, no function exported by M will be inlined into an importing module. The benefit is that modules that import M will need to be recompiled less often (only when M's exports change their type, not when they change their implementation).

-fomit-yields

Default on (yields are *not* inserted)

Tells GHC to omit heap checks when no allocation is being performed. While this improves binary sizes by about 5%, it also means that threads run in tight non-allocating loops will not get preempted in a timely fashion. If it is important to always be able to interrupt such threads, you should turn this optimization off. Consider also recompiling all libraries with this optimization turned off, if you need to guarantee interruptibility.

-fpedantic-bottoms

Default off

Make GHC be more precise about its treatment of bottom (but see also *-fno-state-hack* (page 121)). In particular, stop GHC eta-expanding through a case expression, which is good for performance, but bad if you are using `seq` on partial applications.

-fregs-graph

Default off due to a performance regression bug (#7679)

Only applies in combination with the native code generator. Use the graph colouring register allocator for register allocation in the native code generator. By default, GHC uses a simpler, faster linear register allocator. The downside being that the linear register allocator usually generates worse code.

-fregs-iterative

Default off

Only applies in combination with the native code generator. Use the iterative coalescing graph colouring register allocator for register allocation in the native code generator. This is the same register allocator as the *-fregs-graph* (page 122) one but also enables iterative coalescing during register allocation.

-fsimplifier-phases=⟨n⟩

Default 2

Set the number of phases for the simplifier. Ignored with `-O0`.

-fsimpl-tick-factor=⟨n⟩

Default 100

GHC's optimiser can diverge if you write rewrite rules (*Rewrite rules* (page 589)) that don't terminate, or (less satisfactorily) if you code up recursion through data types (*Bugs in GHC* (page 739)). To avoid making the compiler fall into an infinite loop, the optimiser carries a "tick count" and stops inlining and applying rewrite rules when this count is exceeded. The limit is set as a multiple of the program size, so bigger programs get more ticks. The `-fsimpl-tick-factor` flag lets you change the multiplier. The default is 100; numbers larger than 100 give more ticks, and numbers smaller than 100 give fewer.

If the tick-count expires, GHC summarises what simplifier steps it has done; you can use `-fddump-simpl-stats` to generate a much more detailed list. Usually that identifies the loop quite accurately, because some numbers are very large.

-fdmd-unbox-width=⟨n⟩

Default 3

Boxity analysis optimistically pretends that a function returning a record with at most `-fdmd-unbox-width` fields has only call sites that don't need the box of the returned record. That may in turn allow more argument unboxing to happen. Set to 0 to be completely conservative (which guarantees that no reboxing will happen due to this mechanism).

-fspec-constr

Default off but enabled by *-O2* (page 113).

Turn on call-pattern specialisation; see Call-pattern specialisation for Haskell programs.

This optimisation specializes recursive functions according to their argument "shapes". This is best explained by example so consider:

```
last :: [a] -> a
last [] = error "last"
last (x : []) = x
last (x : xs) = last xs
```

In this code, once we pass the initial check for an empty list we know that in the recursive case this pattern match is redundant. As such `-fspec-constr` will transform the above code to:

```
last :: [a] -> a
last []       = error "last"
last (x : xs) = last' x xs
    where
      last' x []       = x
      last' x (y : ys) = last' y ys
```

As well avoid unnecessary pattern matching it also helps avoid unnecessary allocation. This applies when an argument is strict in the recursive call to itself but not on the initial entry. A strict recursive branch of the function is created similar to the above example.

It is also possible for library writers to instruct GHC to perform call-pattern specialisation extremely aggressively. This is necessary for some highly optimized libraries, where we may want to specialize regardless of the number of specialisations, or the size of the code. As an example, consider a simplified use-case from the `vector` library:

```
import GHC.Types (SPEC(..))

foldl :: (a -> b -> a) -> a -> Stream b -> a
{-# INLINE foldl #-}
foldl f z (Stream step s _) = foldl_loop SPEC z s
  where
    foldl_loop !sPEC z s = case step s of
                            Yield x s' -> foldl_loop sPEC (f z x) s'
                            Skip       -> foldl_loop sPEC z s'
                            Done       -> z
```

Here, after GHC inlines the body of `foldl` to a call site, it will perform call-pattern specialisation very aggressively on `foldl_loop` due to the use of `SPEC` in the argument of the loop body. `SPEC` from `GHC.Types` is specifically recognised by the compiler.

(NB: it is extremely important you use `seq` or a bang pattern on the `SPEC` argument!)

In particular, after inlining this will expose `f` to the loop body directly, allowing heavy specialisation over the recursive cases.

-fspec-constr-keen

Default off

If this flag is on, call-pattern specialisation will specialise a call `(f (Just x))` with an explicit constructor argument, even if the argument is not scrutinised in the body of the function. This is sometimes beneficial; e.g. the argument might be given to some other function that can itself be specialised.

-fspec-constr-count=⟨n⟩

Default 3

Set the maximum number of specialisations that will be created for any one function by the SpecConstr transformation.

-fspec-constr-threshold=⟨n⟩

Default 2000

Set the size threshold for the SpecConstr transformation.

-fspecialise

Default off but enabled by *-O* (page 113).

Specialise each type-class-overloaded function defined in this module for the types at which it is called in this module. If *-fcross-module-specialise* (page 124) is set imported functions that have an INLINABLE pragma (*INLINABLE pragma* (page 612)) will be specialised as well.

-fspecialise-aggressively

Default off

By default only type class methods and methods marked `INLINABLE` or `INLINE` are specialised. This flag will specialise any overloaded function regardless of size if its unfolding is available. This flag is not included in any optimisation level as it can massively increase code size. It can be used in conjunction with *-fexpose-all-unfoldings* (page 117) if you want to ensure all calls are specialised.

-fcross-module-specialise

Default off but enabled by *-O* (page 113).

Specialise `INLINABLE` (*INLINABLE pragma* (page 612)) type-class-overloaded functions imported from other modules for the types at which they are called in this module. Note that specialisation must be enabled (by `-fspecialise`) for this to have any effect.

-fpolymorphic-specialisation

Default off

Warning, this feature is highly experimental and may lead to incorrect runtime results. Use at your own risk (#23469, #23109, #21229, #23445).

Enable specialisation of function calls to known dictionaries with free type variables. The created specialisation will abstract over the type variables free in the dictionary.

-flate-specialise

Default off

Runs another specialisation pass towards the end of the optimisation pipeline. This can catch specialisation opportunities which arose from the previous specialisation pass or other inlining.

You might want to use this if you are you have a type class method which returns a constrained type. For example, a type class where one of the methods implements a traversal.

-fspecialise-incoherents

Default on

Enable specialisation of overloaded functions in cases when the selected instance is incoherent. This makes the choice of instance non-deterministic, so it is only safe to do if there is no observable runtime behaviour difference between potentially unifying instances. Turning this flag off ensures the incoherent instance selection adheres to the algorithm described in *IncoherentInstances* (page 486) at the cost of optimisation opportunities arising from specialisation.

-finline-generics

Default off but enabled by *-O* (page 113).

Since 9.2.1

Annotate methods of derived Generic and Generic1 instances with INLINE[1] pragmas based on heuristics dependent on the size of the data type in question. Improves performance of generics-based algorithms as GHC is able to optimize away intermediate representation more often.

-finline-generics-aggressively

Default off

Since 9.2.1

Annotate methods of all derived Generic and Generic1 instances with INLINE[1] pragmas.

This flag should only be used in modules deriving Generic instances that weren't considered appropriate for INLINE[1] annotations by heuristics of *-finline-generics* (page 124), yet you know that doing so would be beneficial.

When enabled globally it will most likely lead to worse compile times and code size blowup without runtime performance gains.

-fsolve-constant-dicts

Default off but enabled by *-O* (page 113).

When solving constraints, try to eagerly solve super classes using available dictionaries.

For example:

```
class M a b where m :: a -> b

type C a b = (Num a, M a b)

f :: C Int b => b -> Int -> Int
f _ x = x + 1
```

The body of *f* requires a *Num Int* instance. We could solve this constraint from the context because we have *C Int b* and that provides us a solution for *Num Int*. However, we can often produce much better code by directly solving for an available *Num Int* dictionary we might have at hand. This removes potentially many layers of indirection and crucially allows other optimisations to fire as the dictionary will be statically known and selector functions can be inlined.

The optimisation also works for GADTs which bind dictionaries. If we statically know which class dictionary we need then we will solve it directly rather than indirectly using the one passed in at run time.

-fstatic-argument-transformation

Default off

Turn on the static argument transformation, which turns a recursive function into a non-recursive one with a local recursive loop. See Chapter 7 of Andre Santos's PhD thesis.

-fstg-lift-lams

Default off but enabled by *-O2* (page 113).

Enables the late lambda lifting optimisation on the STG intermediate language. This selectively lifts local functions to top-level by converting free variables into function parameters.

-fstg-lift-lams-known

Default off

Allow turning known into unknown calls while performing late lambda lifting. This is deemed non-beneficial, so it’s off by default.

-fstg-lift-lams-non-rec-args

Default 5

Create top-level non-recursive functions with at most <n> parameters while performing late lambda lifting. The default is 5, the number of available parameter registers on x86_64.

-fstg-lift-lams-rec-args

Default 5

Create top-level recursive functions with at most <n> parameters while performing late lambda lifting. The default is 5, the number of available parameter registers on x86_64.

-fstrictness

Default off but enabled by `-O` (page 113).

Turn on demand analysis.

A *Demand* describes an evaluation context of an expression. *Demand analysis* tries to find out what demands a function puts on its arguments when called: If an argument is scrutinised on every code path, the function is strict in that argument and GHC is free to use the more efficient call-by-value calling convention, as well as pass parameters unboxed.

Apart from *strictness analysis*, demand analysis also performs *usage analysis*: Where *strict* translates to “evaluated at least once”, usage analysis asks whether arguments and bindings are “evaluated at most once” or not at all (“evaluated at most zero times”), e.g. *absent*. For the former, GHC may use call-by-name instead of call-by-need, effectively turning thunks into non-memoised functions. For the latter, no code needs to be generated at all: An absent argument can simply be replaced by a dummy value at the call site or omitted altogether.

The worker/wrapper transformation (`-fworker-wrapper` (page 130)) is responsible for exploiting unboxing opportunities and replacing absent arguments by dummies. For arguments that can’t be unboxed, opportunities for call-by-value and call-by-name are exploited in CorePrep when translating to STG.

It’s not only interesting to look at how often a binding is *evaluated*, but also how often a function *is called*. If a function is called at most once, we may freely eta-expand it, even if doing so destroys shared work if the function was called multiple times. This information translates into `OneShotInfo` annotations that the Simplifier acts on.

Notation

So demand analysis is about conservatively inferring lower and upper bounds about how many times something is evaluated/called. We call the “how many times” part a *cardinality*. In the compiler and debug output we differentiate the following cardinality intervals as approximations to cardinality:

Interval	Set of denoted cardinalities	Syntax	Explanation tying syntax to semantics
[1,0]	{}	`B`	Bottom element
[0,0]	{0}	`A`	Absent
[0,1]	{0,1}	`M`	Used at most once ("Maybe")
[0,ω]	{0,1,ω}	`L`	Lazy. Top element, no information, used at least 0, at most many times
[1,1]	{1}	`1`	Strict, used exactly once
[1,ω]	{1,ω}	`S`	Strict, used possibly many times

Note that it's never interesting to differentiate between a cardinality of 2 and 3, or even 4232123. We just approximate the >1 case with ω, standing for "many times".

Apart from the cardinality describing *how often* an argument is evaluated, a demand also carries a *sub-demand*, describing *how deep* something is evaluated beyond a simple `seq`-like evaluation.

This is the full syntax for cardinalities, demands and sub-demands in BNF:

```
card ::= B | A | M | L | 1 | S    semantics as in the table above

d    ::= card sd                  card = how often, sd = how deep
      |  card                     abbreviation: Same as "card card"

sd   ::= card                     polymorphic sub-demand, card at every level
      |  P(d,d,..)                product sub-demand
      |  C(card,sd)               call sub-demand
```

For example, `fst` is strict in its argument, and also in the first component of the argument. It will not evaluate the argument's second component. That is expressed by the demand `1P(1L,A)`. The `P` is for "product sub-demand", which has a *demand* for each product field. The notation `1L` just says "evaluated strictly (`1`), with everything nested inside evaluated according to `L`" – e.g., no information, because that would depend on the evaluation context of the call site of `fst`. The role of `L` in `1L` is that of a *polymorphic* sub-demand, being semantically equivalent to the sub-demand `P(LP(..))`, which we simply abbreviate by the (consequently overloaded) cardinality notation `L`.

For another example, the expression `x + 1` evaluates `x` according to demand `1P(L)`. We have seen single letters stand for cardinalities and polymorphic sub-demands, but what does the single letter `L` mean for a *demand*? Such a single letter demand simply expands to a cardinality and a polymorphic sub-demand of the same letter: E.g. `L` is equivalent to `LL` by expansion of the single letter demand, which is equivalent to `LP(LP(..))`, so `L`s all the way down. It is always clear from context whether we talk about about a cardinality, sub-demand or demand.

Demand signatures

We summarise a function's demand properties in its *demand signature*. This is the general syntax:

```
{x->dx,y->dy,z->dz...}<d1><d2><d3>...<dn>div
        ^              ^   ^   ^      ^   ^
        |              |   |   |      |   |
        |              \---+---+------/   |
        |                  |              |
   demand on free        demand on      divergence
```

(continues on next page)

(continued from previous page)

```
    variables           arguments      information
 (omitted if empty)                    (omitted if
                                     no information)
```

We summarise `fst`'s demand properties in its *demand signature* `<1P(1L,A)>`, which just says "If `fst` is applied to one argument, that argument is evaluated according to `1P(1L,A)`". For another example, the demand signature of `seq` would be `<1A><1L>` and that of `+` would be `<1P(L)><1P(L)>`.

If not omitted, the divergence information can be `b` (surely diverges) or `x` (surely diverges or throws a precise exception). For example, `error` has demand signature `<S>b` and `throwIO` (which is the only way to throw precise exceptions) has demand signature `<_><L><L>x` (leaving out the complicated demand on the `Exception` dictionary).

Call sub-demands

Consider `maybe`:

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe n _ Nothing  = n
maybe _ s (Just a) = s a
```

We give it demand signature `<L><MC(M,L)><1L>`. The `C(M,L)` is a *call sub-demand* that says "Called at most once, where the result is used according to `L`". The expression ``f `seq` f 1`` puts `f` under demand `SC(1,L)` and serves as an example where the upper bound on evaluation cardinality doesn't coincide with that of the call cardinality.

Cardinality is always relative to the enclosing call cardinality, so `g 1 2 + g 3 4` puts `g` under demand `SC(S,C(1,L))`, which says "called multiple times (`S`), but every time it is called with one argument, it is applied exactly once to another argument (`1`)".

-fstrictness-before=⟨n⟩
Run an additional demand analysis before simplifier phase ⟨n⟩.

-funbox-small-strict-fields

Default off but enabled by *-O* (page 113).

This option causes all constructor fields which are marked strict (i.e. "!") and which representation is smaller or equal to the size of a pointer to be unpacked, if possible. It is equivalent to adding an `UNPACK` pragma (see *UNPACK pragma* (page 617)) to every strict constructor field that fulfils the size restriction.

For example, the constructor fields in the following data types

```
data A = A !Int
data B = B !A
newtype C = C B
data D = D !C
```

would all be represented by a single `Int#` (see *Unboxed types and primitive operations* (page 554)) value with `-funbox-small-strict-fields` enabled.

This option is less of a sledgehammer than `-funbox-strict-fields`: it should rarely make things worse. If you use `-funbox-small-strict-fields` to turn on unboxing by default you can disable it for certain constructor fields using the `NOUNPACK` pragma (see *NOUNPACK pragma* (page 618)).

Note that for consistency constructor fields are unpacked on 32-bit platforms as if it we were compiling for a 64-bit target even if fields are larger than a pointer on those platforms.

-funbox-strict-fields

Default off

This option causes all constructor fields which are marked strict (i.e. `!`) to be unpacked if possible. It is equivalent to adding an `UNPACK` pragma to every strict constructor field (see *UNPACK pragma* (page 617)).

This option is a bit of a sledgehammer: it might sometimes make things worse. Selectively unboxing fields by using `UNPACK` pragmas might be better. An alternative is to use `-funbox-strict-fields` to turn on unboxing by default but disable it for certain constructor fields using the `NOUNPACK` pragma (see *NOUNPACK pragma* (page 618)).

Alternatively you can use *-funbox-small-strict-fields* (page 128) to only unbox strict fields which are "small".

-funfolding-creation-threshold=⟨n⟩

Default 750

Governs the maximum size that GHC will allow a function unfolding to be. (An unfolding has a "size" that reflects the cost in terms of "code bloat" of expanding (aka inlining) that unfolding at a call site. A bigger function would be assigned a bigger cost.)

Consequences:

a. nothing larger than this will be inlined (unless it has an `INLINE` pragma)

b. nothing larger than this will be spewed into an interface file.

Increasing this figure is more likely to result in longer compile times than faster code. The *-funfolding-use-threshold=⟨n⟩* (page 129) is more useful.

-funfolding-dict-discount=⟨n⟩

Default 30

How eager should the compiler be to inline dictionaries?

-funfolding-fun-discount=⟨n⟩

Default 60

How eager should the compiler be to inline functions?

-funfolding-keeness-factor=⟨n⟩

This factor was deprecated in GHC 9.0.1. See #15304 for details. Users who need to control inlining should rather consider *-funfolding-use-threshold=⟨n⟩* (page 129).

-funfolding-use-threshold=⟨n⟩

Default 80

This is the magic cut-off figure for unfolding (aka inlining): below this size, a function definition will be unfolded at the call-site, any bigger and it won't. The size computed for a function depends on two things: the actual size of the expression minus any discounts that apply depending on the context into which the expression is to be inlined.

The difference between this and *-funfolding-creation-threshold=⟨n⟩* (page 129) is that this one determines if a function definition will be inlined *at a call site*. The other option determines if a function definition will be kept around at all for potential inlining.

-funfolding-case-threshold=⟨n⟩

Default 2

GHC is in general quite eager to inline small functions. However sometimes these functions will be expanded by more inlining after inlining. Since they are now applied to "interesting" arguments. Even worse, their expanded form might reference again a small function, which will be inlined and expanded afterwards. This can repeat often and lead to explosive growth of programs.

As it happened in #18730.

Starting with GHC 9.0 we will be less eager to inline deep into nested cases. We achieve this by applying a inlining penalty that increases as the nesting gets deeper. However sometimes a specific (maybe quite high!) threshold of nesting is to be expected.

In such cases this flag can be used to ignore the first ⟨n⟩ levels of nesting when computing the penalty.

This flag in combination with *-funfolding-case-scaling=⟨n⟩* (page 130) can be used to break inlining loops without disabling inlining completely. For this purpose a smaller value is more likely to break such loops although often adjusting the scaling is enough and preferably.

-funfolding-case-scaling=⟨n⟩

Default 30

GHC is in general quite eager to inline small functions. However sometimes these functions will be expanded by more inlining after inlining. Since they are now applied to "interesting" arguments. Even worse, their expanded form might reference again a small function, which will be inlined and expanded afterwards. This can repeat often and lead to explosive growth of programs.

As it happened in #18730.

Starting with GHC 9.0 we will be less eager to inline deep into nested cases. We achieve this by applying a inlining penalty that increases as the nesting gets deeper. However sometimes we are ok with inlining a lot in the name of performance.

In such cases this flag can be used to tune how hard we penalize inlining into deeply nested cases beyond the threshold set by *-funfolding-case-threshold=⟨n⟩* (page 129). Cases are only counted against the nesting level if they have more than one alternative.

We use 1/n to scale the penalty. That is a higher value gives a lower penalty.

This can be used to break inlining loops. For this purpose a lower value is recommended. Values in the range 10 <= n <= 20 allow some inlining to take place while still allowing GHC to compile modules containing such inlining loops.

-fworker-wrapper

Default off but enabled by *-O* (page 113).

Enable the worker/wrapper transformation after a demand analysis pass.

Exploits strictness and absence information by unboxing strict arguments and replacing absent fields by dummy values in a wrapper function that will inline in all relevant scenarios and thus expose a specialised, unboxed calling convention of the worker function.

Implied by *-O* (page 113), and by *-fstrictness* (page 126). Disabled by *-fno-strictness* (page 126). Enabling *-fworker-wrapper* (page 130) while demand analysis is disabled (by *-fno-strictness* (page 126)) has no effect.

-fworker-wrapper-cbv

Disabling this flag prevents a W/W split if the only benefit would be call-by-value for some arguments.

Otherwise this exploits strictness information by passing strict value arguments call-by-value to the functions worker. Even for functions who would otherwise not get a worker.

This avoids (potentially repeated) checks for evaluatedness of arguments in the rhs of the worker by pushing this check to the call site. If the argument is statically visible to be a value at the call site the overhead for the check disappears completely.

This can cause slight codesize increases. It will also cause many more functions to get a worker/wrapper split which can play badly with rules (see #20364) which is why it's currently disabled by default. In particular if you depend on rules firing on functions marked as NOINLINE without marking use sites of these functions as INLINE or INLINEABLE then things will break unless this flag is disabled.

While WorkerWrapper is disabled this has no effect.

-fbinary-blob-threshold=⟨n⟩

Default 500000

The native code-generator can either dump binary blobs (e.g. string literals) into the assembly file (by using ".asciz" or ".string" assembler directives) or it can dump them as binary data into a temporary file which is then included by the assembler (using the ".incbin" assembler directive).

This flag sets the size (in bytes) threshold above which the second approach is used. You can disable the second approach entirely by setting the threshold to 0.

5.4 Using Concurrent Haskell

GHC supports Concurrent Haskell by default, without requiring a special option or libraries compiled in a certain way. To get access to the support libraries for Concurrent Haskell, just import Control.Concurrent. More information on Concurrent Haskell is provided in the documentation for that module.

Optionally, the program may be linked with the *-threaded* (page 248) option (see *Options affecting linking* (page 245). This provides two benefits:

- It enables the *-N ⟨x⟩* (page 132) to be used, which allows threads to run in parallel on a multi-processor or multi-core machine. See *Using SMP parallelism* (page 132).
- If a thread makes a foreign call (and the call is not marked `unsafe`), then other Haskell threads in the program will continue to run while the foreign call is in progress. Additionally, `foreign exported` Haskell functions may be called from multiple OS threads simultaneously. See *Multi-threading and the FFI* (page 573).

The following RTS option(s) affect the behaviour of Concurrent Haskell programs:

-C ⟨s⟩

Default 20 milliseconds

Sets the context switch interval to ⟨s⟩ seconds. A context switch will occur at the next heap block allocation after the timer expires (a heap block allocation occurs every 4k of allocation). With `-C0` or `-C`, context switches will occur as often as possible (at every heap block allocation).

5.5 Using SMP parallelism

GHC supports running Haskell programs in parallel on an SMP (symmetric multiprocessor).

There's a fine distinction between *concurrency* and *parallelism*: parallelism is all about making your program run *faster* by making use of multiple processors simultaneously. Concurrency, on the other hand, is a means of abstraction: it is a convenient way to structure a program that must respond to multiple asynchronous events.

However, the two terms are certainly related. By making use of multiple CPUs it is possible to run concurrent threads in parallel, and this is exactly what GHC's SMP parallelism support does. But it is also possible to obtain performance improvements with parallelism on programs that do not use concurrency. This section describes how to use GHC to compile and run parallel programs, in *Parallel and Concurrent* (page 549) we describe the language features that affect parallelism.

5.5.1 Compile-time options for SMP parallelism

In order to make use of multiple CPUs, your program must be linked with the `-threaded` (page 248) option (see *Options affecting linking* (page 245)). Additionally, the following compiler options affect parallelism:

`-feager-blackholing`

Blackholing is the act of marking a thunk (lazy computation) as being under evaluation. It is useful for three reasons: firstly it lets us detect certain kinds of infinite loop (the `NonTermination` exception), secondly it avoids certain kinds of space leak, and thirdly it avoids repeating a computation in a parallel program, because we can tell when a computation is already in progress.

The option `-feager-blackholing` (page 117) causes each thunk to be blackholed as soon as evaluation begins. The default is "lazy blackholing", whereby thunks are only marked as being under evaluation when a thread is paused for some reason. Lazy blackholing is typically more efficient (by 1-2% or so), because most thunks don't need to be blackholed. However, eager blackholing can avoid more repeated computation in a parallel program, and this often turns out to be important for parallelism.

We recommend compiling any code that is intended to be run in parallel with the `-feager-blackholing` (page 117) flag.

5.5.2 RTS options for SMP parallelism

There are two ways to run a program on multiple processors: call Control.Concurrent.setNumCapabilities from your program, or use the RTS `-N ⟨x⟩` (page 132) options.

`-N` ⟨x⟩
`-N`
`-maxN` ⟨x⟩

Use ⟨x⟩ simultaneous threads when running the program.

The runtime manages a set of virtual processors, which we call *capabilities*, the number of which is determined by the `-N` option. Each capability can run one Haskell thread at a time, so the number of capabilities is equal to the number of Haskell threads that can run physically in parallel. A capability is animated by one or more OS threads; the runtime manages a pool of OS threads for each capability, so that if a Haskell thread makes a foreign call (see *Multi-threading and the FFI* (page 573)) another OS thread can take over that capability.

Normally ⟨x⟩ should be chosen to match the number of CPU cores on the machine[1]. For example, on a dual-core machine we would probably use `+RTS -N2 -RTS`.

Omitting ⟨x⟩, i.e. `+RTS -N -RTS`, lets the runtime choose the value of ⟨x⟩ itself based on how many processors are in your machine.

Omitting `-N⟨x⟩` entirely means `-N1`.

With `-maxN⟨x⟩`, i.e. `+RTS -maxN3 -RTS`, the runtime will choose at most (x), also limited by the number of processors on the system. Omitting (x) is an error, if you need a default use option `-N`.

Be careful when using all the processors in your machine: if some of your processors are in use by other programs, this can actually harm performance rather than improve it. Asking GHC to create more capabilities than you have physical threads is almost always a bad idea.

Setting `-N` also has the effect of enabling the parallel garbage collector (see *RTS options to control the garbage collector* (page 184)).

The current value of the `-N` option is available to the Haskell program via `Control.Concurrent.getNumCapabilities`, and it may be changed while the program is running by calling `Control.Concurrent.setNumCapabilities`.

The following options affect the way the runtime schedules threads on CPUs:

`-qa`

Use the OS's affinity facilities to try to pin OS threads to CPU cores.

When this option is enabled, the OS threads for a capability i are bound to the CPU core i using the API provided by the OS for setting thread affinity. e.g. on Linux GHC uses `sched_setaffinity()`.

Depending on your workload and the other activity on the machine, this may or may not result in a performance improvement. We recommend trying it out and measuring the difference.

`-qm`

Disable automatic migration for load balancing. Normally the runtime will automatically try to schedule threads across the available CPUs to make use of idle CPUs; this option disables that behaviour. Note that migration only applies to threads; sparks created by `par` are load-balanced separately by work-stealing.

This option is probably only of use for concurrent programs that explicitly schedule threads onto CPUs with Control.Concurrent.forkOn.

[1] Whether hyperthreading cores should be counted or not is an open question; please feel free to experiment and let us know what results you find.

5.5.3 Hints for using SMP parallelism

Add the *-s [⟨file⟩]* (page 191) RTS option when running the program to see timing stats, which will help to tell you whether your program got faster by using more CPUs or not. If the user time is greater than the elapsed time, then the program used more than one CPU. You should also run the program without *-N ⟨x⟩* (page 132) for comparison.

The output of `+RTS -s` tells you how many "sparks" were created and executed during the run of the program (see *RTS options to control the garbage collector* (page 184)), which will give you an idea how well your `par` annotations are working.

GHC's parallelism support has improved in 6.12.1 as a result of much experimentation and tuning in the runtime system. We'd still be interested to hear how well it works for you, and we're also interested in collecting parallel programs to add to our benchmarking suite.

5.6 Flag reference

This section is a quick-reference for GHC's command-line flags. For each flag, we also list its mode/dynamic status (see *Dynamic and Mode options* (page 67)), and the flag's opposite (if available).

5.6.1 Verbosity options

More details in *Verbosity options* (page 76)

Flag	Description	Type	Reverse
-fabstract-refinement-hole-fits (page 310)	*default: off.* Toggles whether refinements where one or more of the holes are abstract are reported.	dynamic	*-fno-abstract-refinement-hole-* (page 310)
-fdefer-diagnostics (page 80)	Defer and group diagnostic messages by severity	dynamic	
-fdiagnostics-as-json (page 80)	Output diagnostics in Json format specified by JSON schema	dynamic	
-fdiagnostics-color=⟨always|auto|never⟩ (page 80)	Use colors in error messages	dynamic	
-fdiagnostics-show-caret (page 80)	Whether to show snippets of original source code	dynamic	*-fno-diagnostics-show-caret* (page 80)
-ferror-spans (page 81)	Output full span in error messages	dynamic	
-fhide-source-paths (page 76)	hide module source and object paths	dynamic	
-fkeep-going (page 81)	Continue compilation as far as possible on errors	dynamic	

Continued on next page

Table 1 – continued from previous page

Flag	Description	Type	Reverse
-fmax-refinement-hole-fits=⟨n⟩ (page 310)	*default: 6.* Set the maximum number of refinement hole fits for typed holes to display in type error messages.	dynamic	*-fno-max-refinement-hole-fits* (page 310)
-fmax-relevant-binds=⟨n⟩ (page 120)	*default: 6.* Set the maximum number of bindings to display in type error messages.	dynamic	*-fno-max-relevant-binds* (page 120)
-fmax-valid-hole-fits=⟨n⟩ (page 308)	*default: 6.* Set the maximum number of valid hole fits for typed holes to display in type error messages.	dynamic	*-fno-max-valid-hole-fits* (page 308)
-fno-show-valid-hole-fits (page 308)	Disables showing a list of valid hole fits for typed holes in type error messages.	dynamic	
-fno-sort-valid-hole-fits (page 311)	Disables the sorting of the list of valid hole fits for typed holes in type error messages.	dynamic	*-fsort-valid-hole-fits* (page 311)
-fprint-axiom-incomps (page 77)	Display equation incompatibilities in closed type families	dynamic	*-fno-print-axiom-incomps* (page 77)
-fprint-equality-relations (page 78)	Distinguish between equality relations when printing	dynamic	*-fno-print-equality-relations* (page 78)
-fprint-error-index-links=⟨always|auto|never⟩ (page 81)	Whether to emit diagnostic codes as ANSI hyperlinks to the Haskell Error Index.	dynamic	
-fprint-expanded-synonyms (page 78)	In type errors, also print type-synonym-expanded types.	dynamic	*-fno-print-expanded-synonyms* (page 78)
-fprint-explicit-coercions (page 77)	Print coercions in types	dynamic	*-fno-print-explicit-coercions* (page 77)
-fprint-explicit-foralls (page 76)	Print explicit `forall` quantification in types. See also *ExplicitForAll* (page 506)	dynamic	*-fno-print-explicit-foralls* (page 76)
-fprint-explicit-kinds (page 77)	Print explicit kind foralls and kind arguments in types. See also *KindSignatures* (page 511)	dynamic	*-fno-print-explicit-kinds* (page 77)

Continued on next page

Table 1 – continued from previous page

Flag	Description	Type	Reverse
-fprint-explicit-runtime-reps (page 384)	Print `RuntimeRep` and `Levity` variables in types which are runtime-representation polymorphic.	dynamic	*-fno-print-explicit-runtime-rep* (page 384)
-fprint-potential-instances (page 76)	display all available instances in type error messages	dynamic	*-fno-print-potential-instances* (page 76)
-fprint-redundant-promotion-ticks (page 78)	Print redundant *DataKinds* (page 357) promotion ticks	dynamic	*-fno-print-redundant-promotion* (page 78)
-fprint-typechecker-elaboration (page 79)	Print extra information from typechecker.	dynamic	*-fno-print-typechecker-elaborat* (page 79)
-fprint-unicode-syntax (page 76)	Use unicode syntax when printing expressions, types and kinds. See also *UnicodeSyntax* (page 277)	dynamic	*-fno-print-unicode-syntax* (page 76)
-frefinement-level-hole-fits=⟨n⟩ (page 310)	*default: off.* Sets the level of refinement of the refinement hole fits, where level `n` means that hole fits of up to `n` holes will be considered.	dynamic	*-fno-refinement-level-hole-fits* (page 310)
-freverse-errors (page 81)	Output errors in reverse order	dynamic	*-fno-reverse-errors* (page 81)
-fshow-docs-of-hole-fits (page 308)	Toggles whether to show the documentation of the valid hole fits in the output.	dynamic	*-fno-show-docs-of-hole-fits* (page 308)
-fshow-error-context (page 81)	Whether to show textual information about error context	dynamic	*-fno-show-error-context* (page 81)
-fshow-hole-constraints (page 307)	Show constraints when reporting typed holes.	dynamic	
-fshow-hole-matches-of-hole-fits (page 310)	Toggles whether to show the type of the additional holes in refinement hole fits.	dynamic	*-fno-show-hole-matches-of-hole* (page 310)
-fshow-provenance-of-hole-fits (page 308)	Toggles whether to show the provenance of the valid hole fits in the output.	dynamic	*-fno-show-provenance-of-hole-f* (page 308)
-fshow-type-app-of-hole-fits (page 308)	Toggles whether to show the type application of the valid hole fits in the output.	dynamic	*-fno-show-type-app-of-hole-fits* (page 308)

Continued on next page

Table 1 – continued from previous page

Flag	Description	Type	Reverse
-fshow-type-app-vars-of-hole-fits (page 308)	Toggles whether to show what type each quantified variable takes in a valid hole fit.	dynamic	*-fno-show-type-app-vars-of-hole* (page 308)
-fshow-type-of-hole-fits (page 308)	Toggles whether to show the type of the valid hole fits in the output.	dynamic	*-fno-show-type-of-hole-fits* (page 308)
-fsort-by-size-hole-fits (page 311)	Sort valid hole fits by size.	dynamic	*-fno-sort-by-size-hole-fits* (page 311)
-fsort-by-subsumption-hole-fits (page 311)	Sort valid hole fits by subsumption.	dynamic	*-fno-sort-by-subsumption-hole-* (page 311)
-funclutter-valid-hole-fits (page 308)	Unclutter the list of valid hole fits by not showing provenance nor type applications of suggestions.	dynamic	
-Rghc-timing (page 81)	Summarise timing stats for GHC (same as `+RTS -tstderr`).	dynamic	
-v (page 76)	verbose mode (equivalent to `-v3`)	dynamic	
-v⟨n⟩ (page 76)	set verbosity level	dynamic	

5.6.2 Alternative modes of operation

More details in *Modes of operation* (page 68)

Flag	Description	Type	Reverse
--frontend ⟨module⟩ (page 69)	run GHC with the given frontend plugin; see *Frontend plugins* (page 645) for details.	mode	
--help (page 69), *-?* (page 69)	Display help	mode	
--info (page 69)	display information about the compiler	mode	
--interactive (page 68)	Interactive mode - normally used by just running `ghci`; see *Using GHCi* (page 15) for details.	mode	

Continued on next page

Table 2 – continued from previous page

Flag	Description	Type	Reverse
--make (page 68)	Build a multi-module Haskell program, automatically figuring out dependencies. Likely to be much easier, and faster, than using `make`; see *Using ghc --make* (page 71) for details.	mode	
--numeric-version (page 69)	display GHC version (numeric only)	mode	
--print-booter-version (page 69)	display bootstrap compiler version	mode	
--print-build-platform (page 69)	display platform on which GHC was built	mode	
--print-c-compiler-flags (page 69)	C compiler flags used to build GHC	mode	
--print-c-compiler-link-flags (page 69)	C linker flags used to build GHC	mode	
--print-debug-on (page 69)	print whether GHC was built with `-DDEBUG`	mode	
--print-global-package-db (page 69)	display GHC's global package database directory	mode	
--print-have-interpreter (page 69)	display whether GHC was built with interactive support	mode	
--print-have-native-code-generator (page 70)	display whether target platform has NCG support	mode	
--print-host-platform (page 70)	display host platform of GHC	mode	
--print-ld-flags (page 70)	display linker flags used to compile GHC	mode	
--print-leading-underscore (page 70)	display use of leading underscores on symbol names	mode	
--print-libdir (page 70)	display GHC library directory	mode	
--print-object-splitting-supported (page 70)	display whether GHC supports object splitting	mode	
--print-project-git-commit-id (page 70)	display Git commit id GHC is built from	mode	
--print-project-version (page 70)	display GHC version	mode	
--print-rts-ways (page 70)	display which way RTS was built	mode	
--print-stage (page 70)	display `stage` number of GHC	mode	
--print-support-smp (page 70)	display whether GHC was compiled with SMP support	mode	

Continued on next page

Table 2 – continued from previous page

Flag	Description	Type	Reverse
--print-tables-next-to-code (page 71)	display whether GHC was compiled with `--enable-tables-next-to-code`	mode	
--print-target-platform (page 71)	display target platform of GHC	mode	
--print-unregisterised (page 71)	display whether this GHC was built in unregisterised mode	mode	
--run ⟨file⟩ (page 68)	Run a Haskell program.	mode	
--show-iface ⟨file⟩ (page 69)	display the contents of an interface file.	mode	
--show-options (page 69)	display the supported command line options	mode	
--supported-extensions (page 69), *--supported-languages* (page 69)	display the supported language extensions	mode	
--version (page 69), *-V* (page 69)	display GHC version	mode	
-e ⟨expr⟩ (page 68)	Evaluate `expr`; see *Expression evaluation mode* (page 74) for details.	mode	
-M (page 68)	generate dependency information suitable for use in a `Makefile`; see *Dependency generation* (page 216) for details.	mode	
-shared (page 69)	Create a shared object.	mode	

5.6.3 Which phases to run

More details in *Batch compiler mode* (page 75)

Flag	Description	Type	Reverse
--merge-objs (page 68)	Merge a set of objects into a GHCi library.	mode	
-C (page 68)	Stop after generating C (`.hc` file)	mode	
-c (page 68)	Stop after generating object (`.o`) file	mode	
-E (page 68)	Stop after preprocessing (`.hspp` file)	mode	
-F (page 243)	Enable the use of a *pre-processor* (page 243) (set with *-pgmF ⟨cmd⟩* (page 238))	dynamic	

Continued on next page

Table 3 – continued from previous page

Flag	Description	Type	Reverse
-S (page 68)	Stop after generating assembly (`.s` file)	mode	
-x ⟨suffix⟩ (page 75)	Override default behaviour for source files	dynamic	

5.6.4 Redirecting output

More details in *Redirecting the compilation output(s)* (page 202)

Flag	Description	Type	Reverse
-dep-makefile ⟨file⟩ (page 217)	Use ⟨file⟩ as the makefile	dynamic	
-dep-suffix ⟨suffix⟩ (page 217)	Make dependencies that declare that files with suffix `.⟨suf⟩⟨osuf⟩` depend on interface files with suffix `.⟨suf⟩hi`	dynamic	
-dumpdir ⟨dir⟩ (page 203)	redirect dump files	dynamic	
-dynhisuf ⟨suffix⟩ (page 204)	set the suffix to use for dynamic interface files	dynamic	
-dyno ⟨file⟩ (page 202)	set dynamic output filename	dynamic	
-dynohi ⟨file⟩ (page 203)	set the filename in which to put the dynamic interface	dynamic	
-dynosuf ⟨suffix⟩ (page 203)	set the dynamic output file suffix	dynamic	
-hcsuf ⟨suffix⟩ (page 204)	set the suffix to use for intermediate C files	dynamic	
-hidir ⟨dir⟩ (page 203)	set directory for interface files	dynamic	
-hiedir ⟨dir⟩ (page 203)	set directory for extended interface files	dynamic	
-hiesuf ⟨suffix⟩ (page 204)	set the suffix to use for extended interface files	dynamic	
-hisuf ⟨suffix⟩ (page 203)	set the suffix to use for interface files	dynamic	
-o ⟨file⟩ (page 202)	set output filename	dynamic	
-odir ⟨dir⟩ (page 202)	set directory for object files	dynamic	
-ohi ⟨file⟩ (page 203)	set the filename in which to put the interface	dynamic	
-osuf ⟨suffix⟩ (page 203)	set the output file suffix	dynamic	
-outputdir ⟨dir⟩ (page 203)	set output directory	dynamic	

Continued on next page

Table 4 – continued from previous page

Flag	Description	Type	Reverse
-stubdir ⟨dir⟩ (page 203)	redirect FFI stub files	dynamic	

5.6.5 Keeping intermediate files

More details in *Keeping Intermediate Files* (page 204)

Flag	Description	Type	Reverse
-keep-hc-file (page 204), *-keep-hc-files* (page 204)	Retain intermediate `.hc` files.	dynamic	
-keep-hi-files (page 204)	Retain intermediate `.hi` files (the default).	dynamic	*-no-keep-hi-files* (page 204)
-keep-hscpp-file (page 204), *-keep-hscpp-files* (page 204)	Retain intermediate `.hscpp` files.	dynamic	
-keep-llvm-file (page 204), *-keep-llvm-files* (page 204)	Retain intermediate LLVM `.ll` files. Implies *-fllvm* (page 243).	dynamic	
-keep-o-files (page 204)	Retain intermediate `.o` files (the default).	dynamic	*-no-keep-o-files* (page 204)
-keep-s-file (page 204), *-keep-s-files* (page 204)	Retain intermediate `.s` files.	dynamic	
-keep-tmp-files (page 205)	Retain all intermediate temporary files.	dynamic	

5.6.6 Temporary files

More details in *Redirecting temporary files* (page 205)

Flag	Description	Type	Reverse
-tmpdir ⟨dir⟩ (page 205)	set the directory for temporary files	dynamic	

5.6.7 Finding imports

More details in *The search path* (page 201)

Flag	Description	Type	Reverse
-i (page 201)	Empty the import directory list	dynamic	
*-i⟨dir⟩[:⟨dir⟩]** (page 201)	add ⟨dir⟩, ⟨dir2⟩, etc. to import path	dynamic	

5.6.8 Interface file options

More details in *Other options related to interface files* (page 205)

Flag	Description	Type	Reverse
--show-iface ⟨file⟩ (page 69)	See *Modes of operation* (page 68).	mode	
-ddump-hi (page 205)	Dump the new interface to stdout	dynamic	
-ddump-hi-diffs (page 205)	Show the differences vs. the old interface	dynamic	
-ddump-minimal-imports (page 205)	Dump a minimal set of imports	dynamic	

5.6.9 Extended interface file options

More details in *Options related to extended interface files* (page 205)

Flag	Description	Type	Reverse
-fvalidate-ide-info (page 206)	Perform some sanity checks on the extended interface files	dynamic	
-fwrite-ide-info (page 206)	Write out extended interface files	dynamic	

5.6.10 Recompilation checking

More details in *The recompilation checker* (page 206)

Flag	Description	Type	Reverse
-exclude-module=⟨file⟩ (page 217)	Regard `⟨file⟩` as "stable"; i.e., exclude it from having dependencies on it.	dynamic	

Continued on next page

Table 10 – continued from previous page

Flag	Description	Type	Reverse
-fforce-recomp (page 206)	Turn off recompilation checking. This is implied by any `-ddump-X` option when compiling a single file (i.e. when using *-c* (page 68)).	dynamic	*-fno-force-recomp* (page 206)
-fignore-hpc-changes (page 206)	Do not recompile modules just to match changes to HPC flags. This is especially useful for avoiding recompilation when using GHCi, and is enabled by default for GHCi.	dynamic	*-fno-ignore-hpc-changes* (page 206)
-fignore-optim-changes (page 206)	Do not recompile modules just to match changes to optimisation flags. This is especially useful for avoiding recompilation when using GHCi, and is enabled by default for GHCi.	dynamic	*-fno-ignore-optim-changes* (page 206)
-include-cpp-deps (page 217)	Include preprocessor dependencies	dynamic	
-include-pkg-deps (page 217)	Regard modules imported from packages as unstable	dynamic	

5.6.11 Interactive-mode options

More details in *The .ghci and .haskeline files* (page 58)

Flag	Description	Type	Reverse
-fbreak-on-error (page 40)	*Break on uncaught exceptions and errors* (page 40)	dynamic	*-fno-break-on-error* (page 40)
-fbreak-on-exception (page 40)	*Break on any exception thrown* (page 40)	dynamic	*-fno-break-on-exception* (page 40)
-fbreak-points (page 32)	*Insert breakpoints in the GHCi debugger* (page 32)	dynamic	*-fno-break-points* (page 32)
-fghci-hist-size=⟨n⟩ (page 39)	Set the number of entries GHCi keeps for `:history`. See *The GHCi Debugger* (page 32).	dynamic	
-fghci-leak-check (page 43)	(Debugging only) check for space leaks when loading new modules in GHCi.	dynamic	*-fno-ghci-leak-check* (page 43)
-fimplicit-import-qualified (page 26)	Put in scope qualified identifiers for every loaded module	dynamic	*-fno-implicit-import-qualified* (page 26)

Continued on next page

Table 11 – continued from previous page

Flag	Description	Type	Reverse
-flocal-ghci-history (page 42)	Use current directory for the GHCi command history file .ghci-history.	dynamic	*-fno-local-ghci-history* (page 42)
-fno-it (page 28)	No longer set the special variable it.	dynamic	*-fno-no-it* (page 28)
-fprint-bind-result (page 20)	*Turn on printing of binding results in GHCi* (page 20)	dynamic	*-fno-print-bind-result* (page 20)
-fprint-evld-with-show (page 34)	Instruct *:print* (page 51) to use Show instances where possible.	dynamic	
-fshow-loaded-modules (page 16)	Show the names of modules that GHCi loaded after a *:load* (page 50) command.	dynamic	
-ghci-script (page 59)	Read additional .ghci files	dynamic	
-ignore-dot-ghci (page 59)	Disable reading of .ghci files	dynamic	*-no-ignore-dot-ghci* (page 59)
-interactive-print ⟨name⟩ (page 30)	*Select the function to use for printing evaluated expressions in GHCi* (page 30)	dynamic	

5.6.12 Packages

More details in *Packages* (page 219)

Flag	Description	Type	Reverse
-clear-package-db (page 224)	Clear the package db stack.	dynamic	
-distrust ⟨pkg⟩ (page 585)	Expose package ⟨pkg⟩ and set it to be distrusted. See *Safe Haskell* (page 577).	dynamic	
-distrust-all-packages (page 585)	Distrust all packages by default. See *Safe Haskell* (page 577).	dynamic	
-fpackage-trust (page 587)	Enable *Safe Haskell* (page 577) trusted package requirement for trustworthy modules.	dynamic	
-global-package-db (page 225)	Add the global package db to the stack.	dynamic	
-hide-all-packages (page 221)	Hide all packages by default	dynamic	
-hide-package ⟨pkg⟩ (page 222)	Hide package ⟨pkg⟩	dynamic	

Continued on next page

Table 12 – continued from previous page

Flag	Description	Type	Reverse
-ignore-package ⟨pkg⟩ (page 222)	Ignore package ⟨pkg⟩	dynamic	
-no-auto-link-packages (page 222)	Don't automatically link in the base and rts packages.	dynamic	
-no-global-package-db (page 224)	Remove the global package db from the stack.	dynamic	
-no-user-package-db (page 224)	Remove the user's package db from the stack.	dynamic	
-package ⟨pkg⟩ (page 221)	Expose package ⟨pkg⟩	dynamic	
-package-db ⟨file⟩ (page 224)	Add ⟨file⟩ to the package db stack.	dynamic	
-package-env ⟨file⟩|⟨name⟩ (page 226)	Use the specified package environment.	dynamic	
-package-id ⟨unit-id⟩ (page 221)	Expose package by id ⟨unit-id⟩	dynamic	
-this-unit-id ⟨unit-id⟩ (page 222)	Compile to be part of unit (i.e. package) ⟨unit-id⟩	dynamic	
-trust ⟨pkg⟩ (page 585)	Expose package ⟨pkg⟩ and set it to be trusted. See *Safe Haskell* (page 577).	dynamic	
-user-package-db (page 225)	Add the user's package db to the stack.	dynamic	

5.6.13 Language options

Language options can be enabled either by a command-line option `-Xblah`, or by a `{-# LANGUAGE blah #-}` pragma in the file itself. See *Controlling editions and extensions* (page 269).

5.6.14 Warnings

More details in *Warnings and sanity-checking* (page 84)

Flag	Description	Type	Reverse
-fenable-th-splice-warnings (page 534)	Generate warnings for Template Haskell splices	dynamic	*-fno-enable-th-splice-warnings* (page 534)
-fhelpful-errors (page 89)	Make suggestions for misspelled names.	dynamic	*-fno-helpful-errors* (page 89)
-fmax-pmcheck-models=⟨n⟩ (page 95)	soft limit on the number of parallel models the pattern match checker should check a pattern match clause against	dynamic	

Continued on next page

Table 13 – continued from previous page

Flag	Description	Type	Reverse
-fshow-warning-groups (page 87)	show which group an emitted warning belongs to.	dynamic	*-fno-show-warning-groups* (page 87)
-fvia-C (page 237)	use the C code generator	dynamic	
-W (page 86)	enable normal warnings	dynamic	*-Wno-extra* (page 86)
-w (page 86)	disable all warnings	dynamic	
-Wall (page 86)	enable almost all warnings (details in *Warnings and sanity-checking* (page 84))	dynamic	*-Wno-all* (page 86)
-Wall-missed-specialisations (page 89)	warn when specialisation of any overloaded function fails.	dynamic	*-Wno-all-missed-specialisations* (page 89)
-Wall-missed-specializations (page 89)	alias for *-Wall-missed-specialisations* (page 89)	dynamic	*-Wno-all-missed-specializations* (page 89)
-Wambiguous-fields (page 108)	warn about ambiguous field selectors or updates	dynamic	
-Wauto-orphans (page 108)	*(deprecated)* Does nothing	dynamic	
-Wbadly-staged-types (page 111)	warn when type binding is used at the wrong TH stage.	dynamic	*-Wno-badly-staged-types* (page 111)
-Wcompat (page 86)	enable future compatibility warnings (details in *Warnings and sanity-checking* (page 84))	dynamic	*-Wno-compat* (page 86)
-Wcompat-unqualified-imports (page 88)	Report unqualified imports of core libraries which are expected to cause compatibility problems in future releases.	dynamic	*-Wno-compat-unqualified-imports* (page 88)
-Wcpp-undef (page 106)	warn on uses of the *#if* directive on undefined identifiers	dynamic	
-Wdata-kinds-tc (page 111)	warn when an illegal use of a type or kind without *DataKinds* (page 357) is caught by the typechecker	dynamic	*-Wno-data-kinds-tc* (page 111)
-Wdefault (page 85)	enable default flags	dynamic	*-Wno-default* (page 85)
-Wdefaulted-exception-context (page 112)	warn when an Control.Exception.Context.ExceptionContext implicit parameter is defaulted to Control.Exception.Context.emptyExceptionContext.	dynamic	*-Wnop-defaulted-exception-context* (page 112)

Continued on next page

Table 13 – continued from previous page

Flag	Description	Type	Reverse
-Wdeferred-out-of-scope-variables (page 88)	Report warnings when variable out-of-scope errors are *deferred until runtime* (page 414). See *-fdefer-out-of-scope-variables* (page 415).	dynamic	*-Wno-deferred-out-of-scope-variables* (page 88)
-Wdeferred-type-errors (page 88)	Report warnings when *deferred type errors* (page 414) are enabled. This option is enabled by default. See *-fdefer-type-errors* (page 414).	dynamic	*-Wno-deferred-type-errors* (page 88)
-Wdeprecated-flags (page 91)	warn about uses of commandline flags that are deprecated	dynamic	*-Wno-deprecated-flags* (page 91)
-Wdeprecated-type-abstractions (page 110)	warn when type abstractions in constructor patterns are used without enabling *TypeApplications* (page 386)	dynamic	*-Wno-deprecated-type-abstractions* (page 110)
-Wdeprecations (page 90)	warn about uses of functions & types that have DEPRECATED pragmas, or WARNING pragmas with the `deprecated` category.	dynamic	*-Wno-deprecations* (page 90)
-Wderiving-defaults (page 92)	warn about default deriving when using both *DeriveAnyClass* (page 453) and *GeneralizedNewtypeDeriving* (page 447)	dynamic	*-Wno-deriving-defaults* (page 92)
-Wderiving-typeable (page 108)	warn when Typeable is derived	dynamic	*-Wno-deriving-typeable* (page 108)
-Wdodgy-exports (page 92)	warn about dodgy exports	dynamic	*-Wno-dodgy-exports* (page 92)
-Wdodgy-foreign-imports (page 91)	warn about dodgy foreign imports	dynamic	*-Wno-dodgy-foreign-imports* (page 91)
-Wdodgy-imports (page 92)	warn about dodgy imports	dynamic	*-Wno-dodgy-imports* (page 92)
-Wduplicate-constraints (page 92)	warn when a constraint appears duplicated in a type signature	dynamic	*-Wno-duplicate-constraints* (page 92)
-Wduplicate-exports (page 93)	warn when an entity is exported multiple times	dynamic	*-Wno-duplicate-exports* (page 93)

Continued on next page

Table 13 – continued from previous page

Flag	Description	Type	Reverse
-Wempty-enumerations (page 92)	warn about enumerations that are empty	dynamic	*-Wno-empty-enumerations* (page 92)
-Werror (page 87)	make warnings fatal	dynamic	*-Wwarn* (page 87)
-Weverything (page 86)	enable all warnings supported by GHC	dynamic	*-w* (page 86)
-Wextended-warnings (page 89)	warn about uses of functions & types that have WARNING or DEPRECATED pragmas, across all categories	dynamic	*-Wno-extended-warnings* (page 89)
-Wextra (page 86)	alias for *-W* (page 86)	dynamic	*-Wno-extra* (page 86)
-Wforall-identifier (page 108)	*(deprecated)* Does nothing	dynamic	
-Wgadt-mono-local-binds (page 108)	warn when pattern matching on a GADT without MonoLocalBinds	dynamic	*-Wno-gadt-mono-local-binds* (page 108)
-Whi-shadowing (page 93)	*(deprecated)* warn when a `.hi` file in the current directory shadows a library	dynamic	*-Wno-hi-shadowing* (page 93)
-Widentities (page 93)	warn about uses of Prelude numeric conversions that are probably the identity (and hence could be omitted)	dynamic	*-Wno-identities* (page 93)
-Wimplicit-kind-vars (page 93)	*(deprecated)* warn when kind variables are implicitly quantified over.	dynamic	*-Wno-implicit-kind-vars* (page 93)
-Wimplicit-lift (page 94)	warn about implicit `lift` in Template Haskell quotes	dynamic	*-Wno-implicit-lift* (page 94)
-Wimplicit-prelude (page 94)	warn when the Prelude is implicitly imported	dynamic	*-Wno-implicit-prelude* (page 94)
-Wimplicit-rhs-quantification (page 110)	warn when type variables on the RHS of a type synonym are implicitly quantified	dynamic	*-Wno-implicit-rhs-quantification* (page 110)
-Winaccessible-code (page 100)	warn about inaccessible code	dynamic	*-Wno-inaccessible-code* (page 100)
-Wincomplete-export-warnings (page 111)	warn when some but not all of exports for a name are warned about	dynamic	*-Wno-incomplete-export-warnings* (page 111)
-Wincomplete-patterns (page 94)	warn when a pattern match could fail	dynamic	*-Wno-incomplete-patterns* (page 94)

Continued on next page

Table 13 – continued from previous page

Flag	Description	Type	Reverse
-Wincomplete-record-selectors (page 95)	warn when a record selector application could fail	dynamic	*-Wno-incomplete-record-selectors* (page 95)
-Wincomplete-record-updates (page 95)	warn when a record update could fail	dynamic	*-Wno-incomplete-record-updates* (page 95)
-Wincomplete-uni-patterns (page 94)	warn when a pattern match in a lambda expression, pattern binding or a lazy pattern could fail	dynamic	*-Wno-incomplete-uni-patterns* (page 94)
-Winconsistent-flags (page 111)	warn when command line options are inconsistent in some way.	dynamic	*-Wno-inconsistent-flags* (page 111)
-Winferred-safe-imports (page 587)	warn when an explicitly Safe Haskell module imports a Safe-Inferred one	dynamic	*-Wno-inferred-safe-imports* (page 587)
-Winline-rule-shadowing (page 106)	Warn if a rewrite RULE might fail to fire because the function might be inlined before the rule has a chance to fire. See *How rules interact with INLINE/NOINLINE pragmas* (page 593).	dynamic	*-Wno-inline-rule-shadowing* (page 106)
-Winvalid-haddock (page 107)	warn when a Haddock comment occurs in an invalid position	dynamic	*-Wno-invalid-haddock* (page 107)
-Wloopy-superclass-solve (page 109)	*(deprecated)* warn when creating potentially-loopy superclass constraint evidence	dynamic	*-Wno-loopy-superclass-solve* (page 109)
-Wmisplaced-pragmas (page 89)	warn about uses of file header pragmas in the module body	dynamic	*-Wno-misplaced-pragmas* (page 89)
-Wmissed-extra-shared-lib (page 101)	Warn when GHCi can't load a shared lib.	dynamic	*-Wno-missed-extra-shared-lib* (page 101)
-Wmissed-specialisations (page 89)	warn when specialisation of an imported, overloaded function fails.	dynamic	*-Wno-missed-specialisations* (page 89)
-Wmissed-specializations (page 89)	alias for *-Wmissed-specialisations* (page 89)	dynamic	*-Wno-missed-specializations* (page 89)
-Wmissing-deriving-strategies (page 96)	warn when a deriving clause is missing a deriving strategy	dynamic	*-Wno-missing-deriving-strategies* (page 96)
-Wmissing-export-lists (page 96)	warn when a module declaration does not explicitly list all exports	dynamic	*-Wno-missing-export-lists* (page 96)

Continued on next page

Table 13 – continued from previous page

Flag	Description	Type	Reverse
-Wmissing-exported-pattern-synonym-signatures (page 98)	warn about pattern synonyms without signatures, only if they are exported	dynamic	*-Wno-missing-exported-pattern-syn* (page 98)
-Wmissing-exported-signatures (page 97)	warn about top-level functions without signatures, only if they are exported	dynamic	*-Wno-missing-exported-signatures* (page 97)
-Wmissing-exported-sigs (page 97)	*(deprecated)* warn about top-level functions without signatures, only if they are exported. takes precedence over -Wmissing-signatures	dynamic	*-Wno-missing-exported-sigs* (page 97)
-Wmissing-fields (page 96)	warn when fields of a record are uninitialised	dynamic	*-Wno-missing-fields* (page 96)
-Wmissing-home-modules (page 106)	warn when encountering a home module imported, but not listed on the command line. Useful for cabal to ensure GHC won't pick up modules, not listed neither in `exposed-modules`, nor in `other-modules`.	dynamic	*-Wno-missing-home-modules* (page 106)
-Wmissing-import-lists (page 96)	warn when an import declaration does not explicitly list all the names brought into scope	dynamic	*-Wno-missing-import-lists* (page 96)
-Wmissing-kind-signatures (page 98)	warn when type declarations don't have kind signatures nor CUSKs	dynamic	*-Wno-missing-kind-signatures* (page 98)
-Wmissing-local-signatures (page 97)	warn about polymorphic local bindings without signatures	dynamic	*-Wno-missing-local-signatures* (page 97)
-Wmissing-local-sigs (page 97)	*(deprecated)* warn about polymorphic local bindings without signatures	dynamic	*-Wno-missing-local-sigs* (page 97)
-Wmissing-methods (page 97)	warn when class methods are undefined	dynamic	*-Wno-missing-methods* (page 97)
-Wmissing-monadfail-instances (page 91)	*(deprecated)* Warn when a failable pattern is used in a do-block that does not have a `MonadFail` instance.	dynamic	*-Wno-missing-monadfail-instances* (page 91)
-Wmissing-pattern-synonym-signatures (page 97)	warn when pattern synonyms do not have type signatures	dynamic	*-Wno-missing-pattern-synonym-sign* (page 97)

Continued on next page

Table 13 – continued from previous page

Flag	Description	Type	Reverse
-Wmissing-poly-kind-signatures (page 98)	warn when inferred polykinded type or class declaration don't have kind signatures nor CUSKs	dynamic	*-Wno-missing-poly-kind-signatures* (page 98)
-Wmissing-role-annotations (page 110)	warn when type declarations don't have role annotations	dynamic	*-Wno-role-annotations-signatures* (page 110)
-Wmissing-safe-haskell-mode (page 588)	warn when the Safe Haskell mode is not explicitly specified.	dynamic	*-Wno-missing-safe-haskell-mode* (page 588)
-Wmissing-signatures (page 97)	warn about top-level functions without signatures	dynamic	*-Wno-missing-signatures* (page 97)
-Wmissing-space-after-bang (page 108)	*(deprecated)* Does nothing	dynamic	
-Wmonomorphism-restriction (page 101)	warn when the Monomorphism Restriction is applied	dynamic	*-Wno-monomorphism-restriction* (page 101)
-Wname-shadowing (page 98)	warn when names are shadowed	dynamic	*-Wno-name-shadowing* (page 98)
-Wnoncanonical-monad-instances (page 90)	warn when `Applicative` or `Monad` instances have noncanonical definitions of `return`, `pure`, `(>>)`, or `(*>)`. See flag description in *Warnings and sanity-checking* (page 84) for more details.	dynamic	*-Wno-noncanonical-monad-instances* (page 90)
-Wnoncanonical-monadfail-instances (page 90)	*(deprecated)* warn when `Monad` or `MonadFail` instances have noncanonical definitions of `fail`.	dynamic	*-Wno-noncanonical-monadfail-insta* (page 90)
-Wnoncanonical-monoid-instances (page 91)	warn when `Semigroup` or `Monoid` instances have noncanonical definitions of `(<>)` or `mappend`. See flag description in *Warnings and sanity-checking* (page 84) for more details.	dynamic	*-Wno-noncanonical-monoid-instance* (page 91)
-Wnot (page 87)	*(deprecated)* Alias for *-w* (page 86)	dynamic	
-Woperator-whitespace (page 107)	warn on prefix, suffix, and tight infix uses of infix operators	dynamic	*-Wno-operator-whitespace* (page 107)
-Woperator-whitespace-ext-conflict (page 107)	warn on uses of infix operators that would be parsed differently were a particular GHC extension enabled	dynamic	*-Wno-operator-whitespace-ext-conf* (page 107)

Continued on next page

Table 13 – continued from previous page

Flag	Description	Type	Reverse
-Worphans (page 99)	warn when the module contains *orphan instance declarations or rewrite rules* (page 218)	dynamic	*-Wno-orphans* (page 99)
-Woverflowed-literals (page 92)	warn about literals that will overflow their type	dynamic	*-Wno-overflowed-literals* (page 92)
-Woverlapping-patterns (page 99)	warn about overlapping patterns	dynamic	*-Wno-overlapping-patterns* (page 99)
-Wpartial-fields (page 106)	warn when defining a partial record field.	dynamic	*-Wno-partial-fields* (page 106)
-Wpartial-type-signatures (page 88)	warn about holes in partial type signatures when *PartialTypeSignatures* (page 520) is enabled. Not applicable when *PartialTypeSignatures* (page 520) is not enabled, in which case errors are generated for such holes.	dynamic	*-Wno-partial-type-signatures* (page 88)
-Wprepositive-qualified-module (page 88)	Report imports with a leading/prepositive "qualified"	dynamic	*-Wno-prepositive-qualified-module* (page 88)
-Wredundant-bang-patterns (page 104)	Warn about redundant bang patterns.	dynamic	*-Wno-redundant-bang-patterns* (page 104)
-Wredundant-constraints (page 92)	Have the compiler warn about redundant constraints in type signatures.	dynamic	*-Wno-redundant-constraints* (page 92)
-Wredundant-record-wildcards (page 105)	Warn about record wildcard matches when the wildcard binds no patterns.	dynamic	*-Wno-redundant-record-wildcards* (page 105)
-Wredundant-strictness-flags (page 105)	Warn about redundant strictness flags.	dynamic	*-Wno-redundant-strictness-flags* (page 105)
-Wsafe (page 587)	warn if the module being compiled is regarded to be safe.	dynamic	*-Wno-safe* (page 587)
-Wsemigroup (page 91)	*(deprecated)* Warn when a `Monoid` is not `Semigroup`, and on non-`Semigroup` definitions of `(<>)`	dynamic	*-Wno-semigroup* (page 91)

Continued on next page

Table 13 – continued from previous page

Flag	Description	Type	Reverse
-Wsimplifiable-class-constraints (page 101)	Warn about class constraints in a type signature that can be simplified using a top-level instance declaration.	dynamic	*-Wno-simplifiable-class-constraints* (page 101)
-Wstar-binder (page 100)	warn about binding the `(*)` type operator despite *StarIsType* (page 378)	dynamic	*-Wno-star-binder* (page 100)
-Wstar-is-type (page 100)	warn when `*` is used to mean `Data.Kind.Type`	dynamic	*-Wno-star-is-type* (page 100)
-Wtabs (page 101)	warn if there are tabs in the source file	dynamic	*-Wno-tabs* (page 101)
-Wterm-variable-capture (page 109)	warn when an implicitly quantified type variable captures a term's name	dynamic	
-Wtrustworthy-safe (page 587)	warn if the module being compiled is marked as *Trustworthy* (page 586) but it could instead be marked as *Safe* (page 586), a more informative bound.	dynamic	*-Wno-safe* (page 587)
-Wtype-defaults (page 101)	warn when defaulting happens	dynamic	*-Wno-type-defaults* (page 101)
-Wtype-equality-out-of-scope (page 109)	warn when type equality `a ~ b` is used despite being out of scope	dynamic	*-Wno-type-equality-out-of-scope* (page 109)
-Wtype-equality-requires-operators (page 109)	warn when type equality `a ~ b` is used despite being out of scope	dynamic	*-Wno-type-equality-requires-operators* (page 109)
-Wtyped-holes (page 88)	Report warnings when *typed hole* (page 304) errors are *deferred until runtime* (page 414). See *-fdefer-typed-holes* (page 414).	dynamic	*-Wno-typed-holes* (page 88)
-Wunbanged-strict-patterns (page 106)	warn on pattern bind of unlifted variable that is neither bare nor banged	dynamic	*-Wno-unbanged-strict-patterns* (page 106)
-Wunicode-bidirectional-format-characters (page 108)	warn about the usage of unicode bidirectional layout override characters	dynamic	
-Wunrecognised-pragmas (page 89)	warn about uses of pragmas that GHC doesn't recognise	dynamic	*-Wno-unrecognised-pragmas* (page 89)

Continued on next page

Table 13 – continued from previous page

Flag	Description	Type	Reverse
-Wunrecognised-warning-flags (page 88)	throw a warning when an unrecognised `-W...` flag is encountered on the command line.	dynamic	*-Wno-unrecognised-warning-flags* (page 88)
-Wunsafe (page 587)	warn if the module being compiled is regarded to be unsafe. See *Safe Haskell* (page 577)	dynamic	*-Wno-unsafe* (page 587)
-Wunsupported-calling-conventions (page 91)	warn about use of an unsupported calling convention	dynamic	*-Wno-unsupported-calling-conventions* (page 91)
-Wunsupported-llvm-version (page 101)	Warn when using *-fllvm* (page 243) with an unsupported version of LLVM.	dynamic	*-Wno-unsupported-llvm-version* (page 101)
-Wunticked-promoted-constructors (page 102)	warn if promoted constructors are not ticked	dynamic	*-Wno-unticked-promoted-constructors* (page 102)
-Wunused-binds (page 102)	warn about bindings that are unused. Alias for *-Wunused-top-binds* (page 102), *-Wunused-local-binds* (page 102) and *-Wunused-pattern-binds* (page 103)	dynamic	*-Wno-unused-binds* (page 102)
-Wunused-do-bind (page 103)	warn about do bindings that appear to throw away values of types other than `()`	dynamic	*-Wno-unused-do-bind* (page 103)
-Wunused-foralls (page 104)	warn about type variables in user-written `forall`\s that are unused	dynamic	*-Wno-unused-foralls* (page 104)
-Wunused-imports (page 103)	warn about unnecessary imports	dynamic	*-Wno-unused-imports* (page 103)
-Wunused-local-binds (page 102)	warn about local bindings that are unused	dynamic	*-Wno-unused-local-binds* (page 102)
-Wunused-matches (page 103)	warn about variables in patterns that aren't used	dynamic	*-Wno-unused-matches* (page 103)
-Wunused-packages (page 106)	warn when package is requested on command line, but not needed.	dynamic	*-Wno-unused-packages* (page 106)
-Wunused-pattern-binds (page 103)	warn about pattern match bindings that are unused	dynamic	*-Wno-unused-pattern-binds* (page 103)

Continued on next page

Table 13 – continued from previous page

Flag	Description	Type	Reverse
-Wunused-record-wildcards (page 104)	Warn about record wildcard matches when none of the bound variables are used.	dynamic	*-Wno-unused-record-wildcards* (page 104)
-Wunused-top-binds (page 102)	warn about top-level bindings that are unused	dynamic	*-Wno-unused-top-binds* (page 102)
-Wunused-type-patterns (page 104)	warn about unused type variables which arise from patterns in in type family and data family instances	dynamic	*-Wno-unused-type-patterns* (page 104)
-Wwarn (page 87)	make warnings non-fatal	dynamic	*-Werror* (page 87)
-Wwarnings-deprecations (page 90)	warn about uses of functions & types that have DEPRECATED pragmas, or WARNING pragmas with the `deprecated` category. Alias for *-Wdeprecations* (page 90).	dynamic	*-Wno-warnings-deprecations* (page 90)
-Wwrong-do-bind (page 105)	warn about do bindings that appear to throw away monadic values that you should have bound instead	dynamic	*-Wno-wrong-do-bind* (page 105)
-Wx-⟨category⟩ (page 90)	warn about uses of functions & types that have WARNING pragmas with the given category	dynamic	*-Wno-x-⟨category⟩* (page 90)

5.6.15 Optimisation levels

These options are described in more detail in *Optimisation (code improvement)* (page 112).

See *Individual optimisations* (page 156) for a list of optimisations enabled on level 1 and level 2.

Flag	Description	Type	Reverse
-O (page 113), *-O1* (page 113)	Enable level 1 optimisations	dynamic	*-O0* (page 113)
-O0 (page 113)	Disable optimisations (default)	dynamic	
-O2 (page 113)	Enable level 2 optimisations	dynamic	*-O0* (page 113)
-O⟨n⟩ (page 113)	Any -On where n > 2 is the same as -O2.	dynamic	*-O0* (page 113)

5.6.16 Individual optimisations

These options are described in more detail in *-f*: platform-independent flags* (page 113). If a flag is implied by `-O` then it is also implied by `-O2` (unless flag description explicitly says otherwise). If a flag is implied by `-O0` only then the flag is not implied by `-O` and `-O2`.

Flag	Description	Type	Reverse
-fasm-shortcutting (page 114)	Enable shortcutting on assembly. Implied by *-O2* (page 113).	dynamic	*-fno-asm-shortcutting* (page 114)
-fbinary-blob-threshold=⟨n⟩ (page 131)	*default: 500K.* Tweak assembly generator for binary blobs.	dynamic	
-fblock-layout-cfg (page 115)	Use the new cfg based block layout algorithm. Implied by *-O* (page 113).	dynamic	*-fno-block-layout-cfg* (page 115)
-fblock-layout-weightless (page 115)	Ignore cfg weights for code layout.	dynamic	*-fno-block-layout-weightless* (page 115)
-fblock-layout-weights (page 115)	Sets edge weights used by the new code layout algorithm.	dynamic	
-fcall-arity (page 114)	Enable call-arity optimisation. Implied by *-O* (page 113).	dynamic	*-fno-call-arity* (page 114)
-fcase-folding (page 114)	Enable constant folding in case expressions. Implied by *-O* (page 113).	dynamic	*-fno-case-folding* (page 114)
-fcase-merge (page 113)	Enable case-merging. Implied by *-O* (page 113).	dynamic	*-fno-case-merge* (page 113)
-fcmm-control-flow (page 114)	Enable control flow optimisation in the Cmm backend. Implied by *-O* (page 113).	dynamic	*-fno-cmm-control-flow* (page 114)
-fcmm-elim-common-blocks (page 114)	Enable Cmm common block elimination. Implied by *-O* (page 113).	dynamic	*-fno-cmm-elim-common-blocks* (page 114)
-fcmm-sink (page 114)	Enable Cmm sinking. Implied by *-O* (page 113).	dynamic	*-fno-cmm-sink* (page 114)
-fcmm-static-pred (page 114)	Enable static control flow prediction. Implied by *-O* (page 113).	dynamic	*-fno-cmm-static-pred* (page 114)
-fcore-constant-folding (page 113)	Enable constant folding in Core. Implied by *-O* (page 113).	dynamic	*-fno-core-constant-folding* (page 113)
-fcpr-anal (page 115)	Turn on Constructed Product Result analysis. Implied by *-O* (page 113).	dynamic	*-fno-cpr-anal* (page 115)

Continued on next page

Table 15 – continued from previous page

Flag	Description	Type	Reverse
-fcross-module-specialise (page 124)	Turn on specialisation of overloaded functions imported from other modules. Implied by *-O* (page 113).	dynamic	*-fno-cross-module-specialise* (page 124)
-fcse (page 116)	Enable common sub-expression elimination. Implied by *-O* (page 113).	dynamic	*-fno-cse* (page 116)
-fdicts-cheap (page 116)	Make dictionary-valued expressions seem cheap to the optimiser.	dynamic	*-fno-dicts-cheap* (page 116)
-fdicts-strict (page 116)	Make dictionaries strict. Implied by *-O2* (page 113).	dynamic	*-fno-dicts-strict* (page 116)
-fdmd-tx-dict-sel (page 117)	*(deprecated)* Use a special demand transformer for dictionary selectors.	dynamic	*-fno-dmd-tx-dict-sel* (page 117)
-fdmd-unbox-width=⟨n⟩ (page 122)	*default: 3.* Boxity analysis pretends that returned records with this many fields can be unboxed.	dynamic	
-fdo-clever-arg-eta-expansion (page 117)	Enable sophisticated argument eta-expansion. Implied by *-O2* (page 113).	dynamic	*-fno-do-clever-arg-eta-expansion* (page 117)
-fdo-eta-reduction (page 117)	Enable eta-reduction. Always enabled by default.	dynamic	*-fno-do-eta-reduction* (page 117)
-fdo-lambda-eta-expansion (page 117)	Enable lambda eta-expansion. Always enabled by default.	dynamic	*-fno-do-lambda-eta-expansion* (page 117)
-feager-blackholing (page 117)	Turn on *eager blackholing* (page 132)	dynamic	
-fenable-rewrite-rules (page 589)	Switch on all rewrite rules (including rules generated by automatic specialisation of overloaded functions). Implied by *-O* (page 113).	dynamic	*-fno-enable-rewrite-rules* (page 589)
-fexcess-precision (page 117)	Enable excess intermediate precision	dynamic	*-fno-excess-precision* (page 117)
-fexitification (page 114)	Enables exitification optimisation. Implied by *-O* (page 113).	dynamic	*-fno-exitification* (page 114)
-fexpose-all-unfoldings (page 117)	Expose all unfoldings, even for very large or recursive functions.	dynamic	*-fno-expose-all-unfoldings* (page 117)
-ffloat-in (page 118)	Turn on the float-in transformation. Implied by *-O* (page 113).	dynamic	*-fno-float-in* (page 118)

Continued on next page

Table 15 – continued from previous page

Flag	Description	Type	Reverse
-ffull-laziness (page 118)	Turn on full laziness (floating bindings outwards). Implied by -O (page 113).	dynamic	-fno-full-laziness (page 118)
-ffun-to-thunk (page 118)	*(deprecated)* superseded by -ffull-laziness.	dynamic	-fno-fun-to-thunk (page 118)
-fignore-asserts (page 118)	Ignore assertions in the source. Implied by -O (page 113).	dynamic	-fno-ignore-asserts (page 118)
-fignore-interface-pragmas (page 118)	Ignore pragmas in interface files. Implied by -O0 (page 113) only.	dynamic	-fno-ignore-interface-pragmas (page 118)
-finline-generics (page 124)	Annotate methods of derived Generic and Generic1 instances with INLINE[1] pragmas based on heuristics. Implied by -O (page 113).	dynamic	-fno-inline-generics (page 124)
-finline-generics-aggressively (page 125)	Annotate methods of all derived Generic and Generic1 instances with INLINE[1] pragmas.	dynamic	-fno-inline-generics-aggressively (page 125)
-fkeep-auto-rules (page 119)	Keep all "auto" rules, generated by specialisation	dynamic	-fno-keep-auto-rules (page 119)
-flate-dmd-anal (page 119)	Run demand analysis again, at the end of the simplification pipeline	dynamic	-fno-late-dmd-anal (page 119)
-flate-specialise (page 124)	Run a late specialisation pass	dynamic	-fno-late-specialise (page 124)
-fliberate-case (page 119)	Turn on the liberate-case transformation. Implied by -O2 (page 113).	dynamic	-fno-liberate-case (page 119)
-fliberate-case-threshold=⟨n⟩ (page 119)	*default: 2000.* Set the size threshold for the liberate-case transformation to ⟨n⟩	dynamic	-fno-liberate-case-threshold (page 119)
-fllvm-pass-vectors-in-regs (page 119)	*(deprecated)* Does nothing	dynamic	
-flocal-float-out (page 120)	Enable local floating definitions out of let-binds.	dynamic	-fno-local-float-out (page 120)
-flocal-float-out-top-level (page 120)	Enable local floating to float top-level bindings	dynamic	-fno-local-float-out-top-level (page 120)
-floopification (page 119)	Turn saturated self-recursive tail-calls into local jumps in the generated assembly. Implied by -O (page 113).	dynamic	-fno-loopification (page 119)

Continued on next page

Table 15 – continued from previous page

Flag	Description	Type	Reverse
-fmax-inline-alloc-size=⟨n⟩ (page 119)	*default: 128.* Set the maximum size of inline array allocations to ⟨n⟩ bytes (default: 128).	dynamic	
-fmax-inline-memcpy-insns=⟨n⟩ (page 120)	*default: 32.* Inline `memcpy` calls if they would generate no more than ⟨n⟩ pseudo instructions.	dynamic	
-fmax-inline-memset-insns=⟨n⟩ (page 120)	*default: 32.* Inline `memset` calls if they would generate no more than ⟨n⟩ pseudo instructions	dynamic	
-fmax-simplifier-iterations=⟨n⟩ (page 120)	*default: 4.* Set the max iterations for the simplifier.	dynamic	
-fmax-uncovered-patterns=⟨n⟩ (page 120)	*default: 4.* Set the maximum number of patterns to display in warnings about non-exhaustive ones.	dynamic	
-fmax-worker-args=⟨n⟩ (page 121)	*default: 10.* Maximum number of value arguments for a worker.	dynamic	
-fno-opt-coercion (page 121)	Turn off the coercion optimiser	dynamic	
-fno-pre-inlining (page 121)	Turn off pre-inlining	dynamic	
-fno-state-hack (page 121)	Turn off the state hackwhereby any lambda with a real-world state token as argument is considered to be single-entry. Hence OK to inline things inside it.	dynamic	
-fomit-interface-pragmas (page 121)	Don't generate interface pragmas. Implied by -O0 (page 113) only.	dynamic	-fno-omit-interface-pragmas (page 121)
-fomit-yields (page 121)	Omit heap checks when no allocation is being performed.	dynamic	-fno-omit-yields (page 121)
-foptimal-applicative-do (page 283)	Use a slower but better algorithm for ApplicativeDo	dynamic	-fno-optimal-applicative-do (page 283)
-fpedantic-bottoms (page 121)	Make GHC be more precise about its treatment of bottom (but see also -fno-state-hack (page 121)). In particular, GHC will not eta-expand through a case expression.	dynamic	-fno-pedantic-bottoms (page 121)

Continued on next page

Table 15 – continued from previous page

Flag	Description	Type	Reverse
-fpolymorphic-specialisation (page 124)	Allow specialisation to abstract over free type variables	dynamic	-fno-polymorphic-specialisation (page 124)
-fregs-graph (page 122)	Use the graph colouring register allocator for register allocation in the native code generator.	dynamic	-fno-regs-graph (page 122)
-fregs-iterative (page 122)	Use the iterative coalescing graph colouring register allocator in the native code generator.	dynamic	-fno-regs-iterative (page 122)
-fsimpl-tick-factor=⟨n⟩ (page 122)	*default: 100.* Set the percentage factor for simplifier ticks.	dynamic	
-fsimplifier-phases=⟨n⟩ (page 122)	*default: 2.* Set the number of phases for the simplifier. Ignored with -O0 (page 113).	dynamic	
-fsolve-constant-dicts (page 125)	When solving constraints, try to eagerly solve super classes using available dictionaries. Implied by -O (page 113).	dynamic	-fno-solve-constant-dicts (page 125)
-fspec-constr (page 122)	Turn on the SpecConstr transformation. Implied by -O2 (page 113).	dynamic	-fno-spec-constr (page 122)
-fspec-constr-count=⟨n⟩ (page 123)	default: 3.* Set to ⟨n⟩ the maximum number of specialisations that will be created for any one function by the SpecConstr transformation.	dynamic	-fno-spec-constr-count (page 123)
-fspec-constr-keen (page 123)	Specialize a call with an explicit constructor argument, even if the argument is not scrutinised in the body of the function	dynamic	-fno-spec-constr-keen (page 123)
-fspec-constr-threshold=⟨n⟩ (page 123)	*default: 2000.* Set the size threshold for the SpecConstr transformation to ⟨n⟩.	dynamic	-fno-spec-constr-threshold (page 123)
-fspecialise (page 124)	Turn on specialisation of overloaded functions. Implied by -O (page 113).	dynamic	-fno-specialise (page 124)
-fspecialise-aggressively (page 124)	Turn on specialisation of overloaded functions regardless of size, if unfolding is available	dynamic	-fno-specialise-aggressively (page 124)

Continued on next page

Table 15 – continued from previous page

Flag	Description	Type	Reverse
-fspecialise-incoherents (page 124)	Enable specialisation on incoherent instances	dynamic	*-fno-specialise-incoherents* (page 124)
-fstatic-argument-transformation (page 125)	Turn on the static argument transformation.	dynamic	*-fno-static-argument-transformati* (page 125)
-fstg-cse (page 116)	Enable common sub-expression elimination on the STG intermediate language. Implied by *-O* (page 113).	dynamic	*-fno-stg-cse* (page 116)
-fstg-lift-lams (page 125)	Enable late lambda lifting on the STG intermediate language. Implied by *-O2* (page 113).	dynamic	*-fno-stg-lift-lams* (page 125)
-fstg-lift-lams-known (page 126)	Allow turning known into unknown calls while performing late lambda lifting.	dynamic	*-fno-stg-lift-lams-known* (page 126)
-fstg-lift-lams-non-rec-args (page 126)	Create top-level non-recursive functions with at most <n> parameters while performing late lambda lifting.	dynamic	*-fstg-lift-lams-non-rec-args-any* (page 126)
-fstg-lift-lams-rec-args (page 126)	Create top-level recursive functions with at most <n> parameters while performing late lambda lifting.	dynamic	*-fstg-lift-lams-rec-args-any* (page 126)
-fstrictness (page 126)	Turn on demand analysis. Implied by *-O* (page 113). Implies *-fworker-wrapper* (page 130)	dynamic	*-fno-strictness* (page 126)
-fstrictness-before=⟨n⟩ (page 128)	Run an additional demand analysis before simplifier phase ⟨n⟩	dynamic	
-funbox-small-strict-fields (page 128)	Flatten strict constructor fields with a pointer-sized representation. Implied by *-O* (page 113).	dynamic	*-fno-unbox-small-strict-fields* (page 128)
-funbox-strict-fields (page 129)	Flatten strict constructor fields	dynamic	*-fno-unbox-strict-fields* (page 129)
-funfolding-case-scaling=⟨n⟩ (page 130)	*default: 30.* Apply a penalty of (inlining_cost * *1/n*) for each level of case nesting.	dynamic	

Continued on next page

Table 15 – continued from previous page

Flag	Description	Type	Reverse
-funfolding-case-threshold=⟨n⟩ (page 129)	*default: 2.* Reduce inlining for cases nested deeper than n.	dynamic	
-funfolding-creation-threshold=⟨n⟩ (page 129)	*default: 750.* Tweak unfolding settings.	dynamic	
-funfolding-dict-discount=⟨n⟩ (page 129)	*default: 30.* Tweak unfolding settings.	dynamic	
-funfolding-fun-discount=⟨n⟩ (page 129)	*default: 60.* Tweak unfolding settings.	dynamic	
-funfolding-keeness-factor=⟨n⟩ (page 129)	This has been deprecated in GHC 9.0.1.	dynamic	
-funfolding-use-threshold=⟨n⟩ (page 129)	*default: 80.* Tweak unfolding settings.	dynamic	
-fworker-wrapper (page 130)	Enable the worker/wrapper transformation. Implied by -O (page 113) and by -fstrictness (page 126).	dynamic	
-fworker-wrapper-cbv (page 130)	Enable w/w splits for wrappers whos sole purpose is evaluating arguments.	dynamic	

5.6.17 Profiling options

More details in *Profiling* (page 651)

Flag	Description	Type	Reverse
-auto (page 659)	*(deprecated)* Alias for -fprof-auto-exported (page 657)	dynamic	
-auto-all (page 659)	*(deprecated)* Alias for -fprof-auto (page 657)	dynamic	
-caf-all (page 659)	*(deprecated)* Alias for -fprof-cafs (page 658)	dynamic	
-fno-prof-count-entries (page 656)	Do not collect entry counts	dynamic	-fprof-count-entries (page 656)
-fprof-auto (page 657)	Auto-add SCC\ s to all bindings not marked INLINE	dynamic	-fno-prof-auto (page 657)
-fprof-auto-calls (page 657)	Auto-add SCC\ s to all call sites	dynamic	-fno-prof-auto (page 657)
-fprof-auto-exported (page 657)	Auto-add SCC\ s to all exported bindings not marked INLINE (page 609)	dynamic	-fno-prof-auto (page 657)
-fprof-auto-top (page 657)	Auto-add SCC\ s to all top-level bindings not marked INLINE	dynamic	-fno-prof-auto (page 657)

Continued on next page

Table 16 – continued from previous page

Flag	Description	Type	Reverse
-fprof-cafs (page 658)	Auto-add SCC\ s to all CAFs	dynamic	*-fno-prof-cafs* (page 658)
-fprof-callers=⟨name⟩ (page 657)	Auto-add SCC\ s to all call-sites of the named function.	dynamic	
-fprof-late (page 657)	Auto-add SCC\ s to all top level bindings *after* the core pipeline has run.	dynamic	*-fno-prof-late* (page 657)
-fprof-late-inline (page 658)	Auto-add SCC\ s to all top level bindings *after* the optimizer has run and retain them when inlining.	dynamic	*-fno-prof-late-inline* (page 658)
-fprof-late-overloaded (page 658)	Auto-add SCC\ s to all top level overloaded bindings *after* the core pipeline has run.	dynamic	*-fno-prof-late-overloaded* (page 658)
-fprof-late-overloaded-calls (page 658)	Auto-add SCC\ s to all call sites that include dictionary arguments *after* the core pipeline has run.	dynamic	*-fno-prof-late-overloaded-calls* (page 658)
-fprof-manual (page 659)	Process manual SCC annotations.	dynamic	*-fno-prof-manual* (page 659)
-no-auto (page 659)	*(deprecated)* Alias for *-fno-prof-auto* (page 657)	dynamic	
-no-auto-all (page 659)	*(deprecated)* Alias for *-fno-prof-auto* (page 657)	dynamic	
-no-caf-all (page 659)	*(deprecated)* Alias for *-fno-prof-cafs* (page 658)	dynamic	
-prof (page 656)	Turn on profiling	dynamic	
-ticky (page 677)	Turn on *ticky-ticky profiling* (page 677)	dynamic	
-ticky-allocd (page 678)	Track the number of times each closure type is allocated.	dynamic	
-ticky-ap-thunk (page 678)	Don't use standard AP thunks on order to get more reliable entry counters.	dynamic	
-ticky-dyn-thunk (page 678)	Track allocations of dynamic thunks	dynamic	
-ticky-LNE (page 678)	Treat join point binders similar to thunks/functions.	dynamic	

Continued on next page

Table 16 – continued from previous page

Flag	Description	Type	Reverse
-ticky-tag-checks (page 678)	Emit dummy ticky counters to record how many tag-inference checks tag inference avoided.	dynamic	

5.6.18 Program coverage options

More details in *Observing Code Coverage* (page 672)

Flag	Description	Type	Reverse
-fhpc (page 674)	Turn on Haskell program coverage instrumentation	dynamic	
-hpcdir⟨dir⟩ (page 674)	Set the directory where GHC places `.mix` files.	dynamic	

5.6.19 C pre-processor options

More details in *Options affecting the C pre-processor* (page 240)

Flag	Description	Type	Reverse
-cpp (page 240)	Run the C pre-processor on Haskell source files	dynamic	
-D⟨symbol⟩[=⟨value⟩] (page 240)	Define a symbol in the C pre-processor	dynamic	*-U⟨symbol⟩* (page 240)
-I⟨dir⟩ (page 240)	Add ⟨dir⟩ to the directory search list for `#include` files	dynamic	
-U⟨symbol⟩ (page 240)	Undefine a symbol in the C pre-processor	dynamic	

5.6.20 Code generation options

More details in *Options affecting code generation* (page 243)

Flag	Description	Type	Reverse
-dynamic-too (page 245)	Build dynamic object files *as well as* static object files during compilation	dynamic	
-fasm (page 243)	Use the *native code generator* (page 236)	dynamic	*-fllvm* (page 243)
-fbyte-code (page 244)	Generate byte-code	dynamic	
-fbyte-code-and-object-code (page 244)	Generate object code and byte-code	dynamic	

Continued on next page

Table 19 – continued from previous page

Flag	Description	Type	Reverse
-fexpose-internal-symbols (page 245)	Produce symbols for all functions, including internal functions.	dynamic	
-fexternal-dynamic-refs (page 244)	Generate code for linking against dynamic libraries	dynamic	
-fllvm (page 243)	Compile using the *LLVM code generator* (page 236)	dynamic	*-fasm* (page 243)
-fno-code (page 243)	Omit code generation	dynamic	
-fobject-code (page 244)	Generate object code	dynamic	
-fPIC (page 244)	Generate position-independent code (where available)	dynamic	
-fPIE (page 244)	Generate code for a position-independent executable (where available)	dynamic	
-fprefer-byte-code (page 245)	Use byte-code if it is available to evaluate TH splices	dynamic	
-fwrite-if-simplified-core (page 244)	Write an interface file containing the simplified core of the module.	dynamic	
-fwrite-interface (page 244)	Always write interface files	dynamic	

5.6.21 Linking options

More details in *Options affecting linking* (page 245)

Flag	Description	Type	Reverse
-c (page 68)	Stop after generating object (`.o`) file	mode	
-debug (page 248)	Use the debugging runtime	dynamic	
-dylib-install-name ⟨path⟩ (page 250)	Set the install name (via `-install_name` passed to Apple's linker), specifying the full install path of the library file. Any libraries or executables that link with it later will pick up that path as their runtime search location for it. (Darwin/OS X only)	dynamic	
-dynamic (page 246)	Build dynamically-linked object files and executables	dynamic	
-dynload (page 247)	Selects one of a number of modes for finding shared libraries at runtime.	dynamic	
-eventlog (page 248)	Enable runtime event tracing	dynamic	

Continued on next page

Table 20 – continued from previous page

Flag	Description	Type	Reverse
-fcompact-unwind (page 251)	Instruct the linker to produce a *__compact_unwind* section.	dynamic	
-fkeep-cafs (page 251)	Do not garbage-collect CAFs (top-level expressions) at runtime	dynamic	
-flink-rts (page 247)	Link the runtime when generating a shared or static library	dynamic	
-fno-embed-manifest (page 250)	Do not embed the manifest in the executable (Windows only)	dynamic	
-fno-gen-manifest (page 249)	Do not generate a manifest file (Windows only)	dynamic	
-fno-shared-implib (page 250)	Don't generate an import library for a DLL (Windows only)	dynamic	
-framework ⟨name⟩ (page 246)	On Darwin/OS X/iOS only, link in the framework ⟨name⟩. This option corresponds to the `-framework` option for Apple's Linker.	dynamic	
-framework-path ⟨dir⟩ (page 246)	On Darwin/OS X/iOS only, add ⟨dir⟩ to the list of directories searched for frameworks. This option corresponds to the `-F` option for Apple's Linker.	dynamic	
-fsplit-sections (page 246), *-split-sections* (page 246)	Split sections for link-time dead-code stripping	dynamic	*-fno-split-sections* (page 246)
-fuse-rpaths (page 246)	Set the rpath based on -L flags	dynamic	
-fwhole-archive-hs-libs (page 250)	When linking a binary executable, this inserts the flag `-Wl,--whole-archive` before any `-l` flags for Haskell libraries, and `-Wl,--no-whole-archive` afterwards	dynamic	
-L ⟨dir⟩ (page 246)	Add ⟨dir⟩ to the list of directories searched for libraries	dynamic	
-l ⟨lib⟩ (page 245)	Link in library ⟨lib⟩	dynamic	
-main-is ⟨thing⟩ (page 247)	Set main module and function	dynamic	
-no-hs-main (page 247)	Don't assume this program contains `main`	dynamic	
-no-pie (page 251)	Don't instruct the linker to produce a position-independent executable.	dynamic	*-pie* (page 251)
-no-rtsopts-suggestions (page 249)	Don't print RTS suggestions about linking with *-rtsopts[=⟨none|some|all|ignore|ignoreAll⟩]* (page 248).	dynamic	
-package ⟨name⟩ (page 246)	Expose package ⟨pkg⟩	dynamic	
-pie (page 251)	Instruct the linker to produce a position-independent executable.	dynamic	*-no-pie* (page 251)

Continued on next page

Table 20 – continued from previous page

Flag	Description	Type	Reverse
-rdynamic (page 250)	This instructs the linker to add all symbols, not only used ones, to the dynamic symbol table. Currently Linux and Windows/MinGW32 only. This is equivalent to using `-optl -rdynamic` on Linux, and `-optl -export-all-symbols` on Windows.	dynamic	
-rtsopts[=⟨none|some|all|ignore|ignoreAll⟩] (page 248)	Control whether the RTS behaviour can be tweaked via command-line flags and the `GHCRTS` environment variable. Using `none` means no RTS flags can be given; `some` means only a minimum of safe options can be given (the default); `all` (or no argument at all) means that all RTS flags are permitted; `ignore` means RTS flags can be given, but are treated as regular arguments and passed to the Haskell program as arguments; `ignoreAll` is the same as `ignore`, but `GHCRTS` is also ignored. `-rtsopts` does not affect `-with-rtsopts` behavior; flags passed via `-with-rtsopts` are used regardless of `-rtsopts`.	dynamic	
-shared (page 69)	Generate a shared library (as opposed to an executable)	dynamic	
-single-threaded (page 248)	Use the single-threaded runtime	dynamic	*-threaded* (page 248)
-static (page 246)	Use static Haskell libraries	dynamic	
-staticlib (page 246)	Generate a standalone static library (as opposed to an executable). This is useful when cross compiling. The library together with all its dependencies ends up in in a single static library that can be linked against.	dynamic	
-threaded (page 248)	Use the threaded runtime	dynamic	*-single-threaded* (page 248)
-with-rtsopts=⟨opts⟩ (page 249)	Set the default RTS options to ⟨opts⟩.	dynamic	

5.6.22 Plugin options

More details in *Compiler Plugins* (page 625)

Flag	Description	Type	Reverse
`-fclear-plugins` (page 626)	Clear the list of active plugins	dynamic	
`-fplugin-library=⟨file-path⟩;⟨unit-id⟩;⟨module⟩;⟨args⟩` (page 626)	Load a pre-compiled static plugin from an external library	dynamic	
`-fplugin-opt=⟨module⟩:⟨args⟩` (page 625)	Give arguments to a plugin module; module must be specified with `-fplugin=⟨module⟩` (page 625)	dynamic	
`-fplugin-trustworthy` (page 626)	Trust the used plugins and no longer mark the compiled module as unsafe	dynamic	
`-fplugin=⟨module⟩` (page 625)	Load a plugin exported by a given module	dynamic	
`-hide-all-plugin-packages` (page 627)	Hide all packages for plugins by default	dynamic	
`-plugin-package ⟨pkg⟩` (page 627)	Expose ⟨pkg⟩ for plugins	dynamic	
`-plugin-package-id ⟨pkg-id⟩` (page 627)	Expose ⟨pkg-id⟩ for plugins	dynamic	

5.6.23 Replacing phases

More details in *Replacing the program for one or more phases* (page 238)

Flag	Description	Type	Reverse
`-pgma ⟨cmd⟩` (page 238)	Use ⟨cmd⟩ as the assembler	dynamic	
`-pgmc ⟨cmd⟩` (page 238)	Use ⟨cmd⟩ as the C compiler	dynamic	
`-pgmcxx ⟨cmd⟩` (page 238)	Use ⟨cmd⟩ as the C++ compiler	dynamic	
`-pgmF ⟨cmd⟩` (page 238)	Use ⟨cmd⟩ as the pre-processor (with `-F` (page 243) only)	dynamic	
`-pgmi ⟨cmd⟩` (page 238)	Use ⟨cmd⟩ as the external interpreter command.	dynamic	
`-pgminstall_name_tool ⟨cmd⟩` (page 238)	Use ⟨cmd⟩ as the program to inject `runpath` into mach-o dylibs on macOS	dynamic	
`-pgmL ⟨cmd⟩` (page 238)	Use ⟨cmd⟩ as the literate pre-processor	dynamic	
`-pgml ⟨cmd⟩` (page 238)	Use ⟨cmd⟩ as the linker	dynamic	

Continued on next page

Table 22 – continued from previous page

Flag	Description	Type	Reverse
-pgmlas ⟨cmd⟩ (page 238)	Use ⟨cmd⟩ as the LLVM assembler	dynamic	
-pgmlc ⟨cmd⟩ (page 238)	Use ⟨cmd⟩ as the LLVM compiler	dynamic	
-pgmlm ⟨cmd⟩ (page 238)	Use ⟨cmd⟩ as the linker when merging object files	dynamic	
-pgmlo ⟨cmd⟩ (page 238)	Use ⟨cmd⟩ as the LLVM optimiser	dynamic	
-pgmotool ⟨cmd⟩ (page 238)	Use ⟨cmd⟩ as the program to inspect mach-o dylibs on macOS	dynamic	
-pgmP ⟨cmd⟩ (page 238)	Use ⟨cmd⟩ as the C pre-processor (with *-cpp* (page 240) only)	dynamic	
-pgms ⟨cmd⟩ (page 238)	Use ⟨cmd⟩ as the splitter	dynamic	
-pgmwindres ⟨cmd⟩ (page 238)	Use ⟨cmd⟩ as the program for embedding manifests on Windows.	dynamic	

5.6.24 Forcing options to particular phases

More details in *Forcing options to a particular phase* (page 239)

Flag	Description	Type	Reverse
-opta ⟨option⟩ (page 239)	pass ⟨option⟩ to the assembler	dynamic	
-optc ⟨option⟩ (page 239)	pass ⟨option⟩ to the C compiler	dynamic	
-optcxx ⟨option⟩ (page 239)	pass ⟨option⟩ to the C++ compiler	dynamic	
-optF ⟨option⟩ (page 239)	pass ⟨option⟩ to the custom pre-processor	dynamic	
-opti ⟨option⟩ (page 239)	pass ⟨option⟩ to the interpreter sub-process.	dynamic	
-optL ⟨option⟩ (page 239)	pass ⟨option⟩ to the literate pre-processor	dynamic	
-optl ⟨option⟩ (page 239)	pass ⟨option⟩ to the linker	dynamic	
-optlas ⟨option⟩ (page 239)	pass ⟨option⟩ to the LLVM assembler	dynamic	
-optlc ⟨option⟩ (page 239)	pass ⟨option⟩ to the LLVM compiler	dynamic	
-optlm ⟨option⟩ (page 239)	pass ⟨option⟩ to the linker when merging object files.	dynamic	
-optlo ⟨option⟩ (page 239)	pass ⟨option⟩ to the LLVM optimiser	dynamic	
-optP ⟨option⟩ (page 239)	pass ⟨option⟩ to cpp (with *-cpp* (page 240) only)	dynamic	

Continued on next page

Table 23 – continued from previous page

Flag	Description	Type	Reverse
-optwindres ⟨option⟩ (page 239)	pass ⟨option⟩ to `windres`.	dynamic	
-pgmc-supports-no-pie (page 239)	*(deprecated)* Indicate that the linker supports `-no-pie`	dynamic	
-pgml-supports-no-pie (page 239)	Indicate that the linker supports `-no-pie`	dynamic	

5.6.25 Platform-specific options

More details in *Platform-specific Flags* (page 82)

Flag	Description	Type	Reverse
-mavx (page 82)	(x86 only) Enable support for AVX SIMD extensions	dynamic	
-mavx2 (page 82)	(x86 only) Enable support for AVX2 SIMD extensions	dynamic	
-mavx512cd (page 82)	(x86 only) Enable support for AVX512-CD SIMD extensions	dynamic	
-mavx512er (page 82)	(x86 only) Enable support for AVX512-ER SIMD extensions	dynamic	
-mavx512f (page 82)	(x86 only) Enable support for AVX512-F SIMD extensions	dynamic	
-mavx512pf (page 82)	(x86 only) Enable support for AVX512-PF SIMD extensions	dynamic	
-mbmi (page 83)	(x86 only) Use BMI1 for bit manipulation operations	dynamic	
-mbmi2 (page 83)	(x86 only) Use BMI2 for bit manipulation operations	dynamic	
-mfma (page 83)	Use native FMA instructions for fused multiply-add floating-point operations	dynamic	
-msse (page 82)	(x86 only) Use SSE for floating-point operations	dynamic	
-msse2 (page 82)	(x86 only) Use SSE2 for floating-point operations	dynamic	
-msse3 (page 83)	(x86 only) Use SSE3 for floating-point operations	dynamic	
-msse4 (page 83)	(x86 only) Use SSE4 for floating-point operations	dynamic	
-msse4.2 (page 83)	(x86 only) Use SSE4.2 for floating-point operations	dynamic	

5.6.26 Compiler debugging options

More details in *Debugging the compiler* (page 255)

Flag	Description	Type	Reverse
`-dasm-lint` (page 265)	ASM pass sanity checking	dynamic	
`-dcmm-lint` (page 265)	C-\- pass sanity checking	dynamic	
`-dcore-lint` (page 265)	Turn on internal sanity checking	dynamic	
`-ddisable-js-c-sources` (page 263)	Disable the link with C sources compiled to JavaScript	dynamic	
`-ddisable-js-minifier` (page 263)	Generate pretty-printed JavaScript code instead of minified (compacted) code.	dynamic	
`-ddump-asm` (page 262)	Dump final assembly	dynamic	
`-ddump-asm-conflicts` (page 262)	Dump register conflicts from the register allocator.	dynamic	
`-ddump-asm-liveness` (page 262)	Dump assembly augmented with register liveness	dynamic	
`-ddump-asm-native` (page 262)	Dump initial assembly	dynamic	
`-ddump-asm-regalloc` (page 262)	Dump the result of register allocation	dynamic	
`-ddump-asm-regalloc-stages` (page 262)	Dump the build/spill stages of the `-fregs-graph` (page 122) register allocator.	dynamic	
`-ddump-asm-stats` (page 262)	Dump statistics from the register allocator.	dynamic	
`-ddump-bcos` (page 263)	Dump interpreter byte code	dynamic	
`-ddump-c-backend` (page 262)	Dump C code produced by the C (unregisterised) backend.	dynamic	
`-ddump-call-arity` (page 258)	Dump output of the call arity analysis pass.	dynamic	
`-ddump-cfg-weights` (page 261)	Dump the assumed weights of the CFG.	dynamic	
`-ddump-cmm` (page 261)	Dump the final C-\- output	dynamic	
`-ddump-cmm-caf` (page 261)	Dump the results of the C-\- CAF analysis pass.	dynamic	
`-ddump-cmm-cbe` (page 261)	Dump the results of common block elimination	dynamic	
`-ddump-cmm-cfg` (page 261)	Dump the results of the C-\- control flow optimisation pass.	dynamic	
`-ddump-cmm-cps` (page 261)	Dump the results of the CPS pass	dynamic	
`-ddump-cmm-from-stg` (page 261)	Dump STG-to-C-\- output	dynamic	

Continued on next page

Table 25 – continued from previous page

Flag	Description	Type	Reverse
-ddump-cmm-info (page 261)	Dump the results of the C-\- info table augmentation pass.	dynamic	
-ddump-cmm-opt (page 262)	Dump the results of C-\- to C-\- optimising passes	dynamic	
-ddump-cmm-proc (page 261)	Dump the results of proc-point analysis	dynamic	
-ddump-cmm-procmap (page 261)	Dump the results of the C-\- proc-point map pass.	dynamic	
-ddump-cmm-raw (page 261)	Dump raw C-\-	dynamic	
-ddump-cmm-sink (page 261)	Dump the results of the C-\- sinking pass.	dynamic	
-ddump-cmm-sp (page 261)	Dump the results of the C-\- stack layout pass.	dynamic	
-ddump-cmm-split (page 261)	Dump the results of the C-\- proc-point splitting pass.	dynamic	
-ddump-cmm-switch (page 261)	Dump the results of switch lowering passes	dynamic	
-ddump-cmm-thread-sanitizer (page 261)	Dump the results of the C-\- ThreadSanitizer elaboration pass.	dynamic	
-ddump-cmm-verbose (page 260)	Write output from main C-\- pipeline passes to files	dynamic	
-ddump-cmm-verbose-by-proc (page 260)	Show output from main C-\- pipeline passes (grouped by proc)	dynamic	
-ddump-core-stats (page 258)	Print a one-line summary of the size of the Core program at the end of the optimisation pipeline	dynamic	
-ddump-cpr-signatures (page 259)	Dump CPR signatures	dynamic	
-ddump-cpranal (page 259)	Dump CPR analysis output	dynamic	
-ddump-cs-trace (page 257)	Trace constraint solver	dynamic	
-ddump-cse (page 259)	Dump CSE output	dynamic	
-ddump-debug (page 263)	Dump generated DWARF debug information	dynamic	
-ddump-deriv (page 257)	Dump deriving output	dynamic	
-ddump-dmd-signatures (page 259)	Dump top-level demand signatures	dynamic	
-ddump-dmdanal (page 259)	Dump demand analysis output	dynamic	
-ddump-ds (page 258), *-ddump-ds-preopt* (page 258)	Dump desugarer output.	dynamic	
-ddump-ec-trace (page 257)	Trace exhaustiveness checker	dynamic	
-ddump-exitify (page 258)	Dump output of the exitification pass.	dynamic	

Continued on next page

Table 25 – continued from previous page

Flag	Description	Type	Reverse
-ddump-faststrings (page 256)	Dump the whole FastString table when finished	dynamic	
-ddump-file-prefix=⟨str⟩ (page 255)	Set the prefix of the filenames used for debugging output.	dynamic	
-ddump-float-in (page 259)	Dump float in output	dynamic	
-ddump-foreign (page 263)	Dump `foreign export` stubs	dynamic	
-ddump-full-laziness (page 259), -ddump-float-out (page 259)	Dump full laziness pass output	dynamic	
-ddump-hie (page 257)	Dump the hie file syntax tree	dynamic	
-ddump-hpc (page 263)	An alias for -ddump-ticked (page 263).	dynamic	
-ddump-if-trace (page 256)	Trace interface files	dynamic	
-ddump-inlinings (page 259)	Dump inlinings performed by the simplifier.	dynamic	
-ddump-js (page 262)	Dump final JavaScript code	dynamic	
-ddump-json (page 256)	*(deprecated)* Use -fdiagnostics-as-json (page 80) instead	dynamic	
-ddump-late-cc (page 260)	Dump core with late cost centres added	dynamic	
-ddump-liberate-case (page 259)	Dump liberate case output	dynamic	
-ddump-llvm (page 262)	Dump LLVM intermediate code.	dynamic	
-ddump-mod-map (page 263)	Dump the state of the module mapping database.	dynamic	
-ddump-occur-anal (page 260)	Dump occurrence analysis output	dynamic	
-ddump-opt-cmm (page 262)	Dump the results of C-\- to C-\- optimising passes	dynamic	
-ddump-parsed (page 256)	Dump parse tree	dynamic	
-ddump-parsed-ast (page 256)	Dump parser output as a syntax tree	dynamic	
-ddump-prep (page 260)	Dump prepared core	dynamic	
-ddump-rn (page 257)	Dump renamer output	dynamic	
-ddump-rn-ast (page 257)	Dump renamer output as a syntax tree	dynamic	
-ddump-rn-stats (page 257)	Renamer stats	dynamic	
-ddump-rn-trace (page 257)	Trace renamer	dynamic	

Continued on next page

Table 25 – continued from previous page

Flag	Description	Type	Reverse
-ddump-rtti (page 263)	Trace runtime type inference	dynamic	
-ddump-rule-firings (page 258)	Dump rule firing info	dynamic	
-ddump-rule-rewrites (page 258)	Dump detailed rule firing info	dynamic	
-ddump-rules (page 258)	Dump rewrite rules	dynamic	
-ddump-simpl (page 259)	Dump final simplifier output	dynamic	
-ddump-simpl-iterations (page 258)	Dump output from each simplifier iteration	dynamic	
-ddump-simpl-stats (page 258)	Dump simplifier stats	dynamic	
-ddump-simpl-trace (page 258)	Dump trace messages in simplifier	dynamic	
-ddump-spec (page 258)	Dump specialiser output	dynamic	
-ddump-spec-constr (page 258)	Dump specialiser output from SpecConstr	dynamic	
-ddump-splices (page 257)	Dump TH spliced expressions, and what they evaluate to	dynamic	
-ddump-static-argument-transformation (page 260)	Dump static argument transformation output	dynamic	
-ddump-stg (page 260)	*(deprecated)* Alias for *-ddump-stg-from-core* (page 260)	dynamic	
-ddump-stg-cg (page 260)	Show output after Stg2Stg	dynamic	
-ddump-stg-final (page 260)	Show output of last STG pass.	dynamic	
-ddump-stg-from-core (page 260)	Show CoreToStg output	dynamic	
-ddump-stg-tags (page 260)	Show output of the tag inference pass.	dynamic	
-ddump-stg-unarised (page 260)	Show unarised STG	dynamic	
-ddump-str-signatures (page 259)	*(deprecated)* Alias for *-ddump-dmd-signatures* (page 259)	dynamic	
-ddump-stranal (page 259)	*(deprecated)* Alias for *-ddump-dmdanal* (page 259)	dynamic	
-ddump-tc (page 257)	Dump typechecker output	dynamic	
-ddump-tc-ast (page 257)	Dump typechecker output as a syntax tree	dynamic	
-ddump-tc-trace (page 257)	Trace typechecker	dynamic	
-ddump-ticked (page 263)	Dump the code instrumented by HPC (*Observing Code Coverage* (page 672)).	dynamic	

Continued on next page

Table 25 – continued from previous page

Flag	Description	Type	Reverse
-ddump-timings (page 256)	Dump per-pass timing and allocation statistics	dynamic	
-ddump-to-file (page 255)	Dump to files instead of stdout	dynamic	
-ddump-types (page 257)	Dump type signatures	dynamic	
-ddump-verbose-inlinings (page 259)	Dump all considered inlinings	dynamic	
-ddump-view-pattern-commoning (page 260)	Dump commoned view patterns	dynamic	
-ddump-worker-wrapper (page 260)	Dump worker-wrapper output	dynamic	
-dfaststring-stats (page 256)	Show statistics for fast string usage when finished	dynamic	
-dhex-word-literals (page 264)	Print values of type *Word#* in hexadecimal.	dynamic	
-dinitial-unique=⟨s⟩ (page 267)	Start `UniqSupply` allocation from ⟨s⟩.	dynamic	
-dinline-check=⟨str⟩ (page 259)	Dump information about inlining decisions	dynamic	
-dipe-stats (page 256)	Show statistics about IPE information	dynamic	
-dkeep-comments (page 256)	Include comments in the parser. Useful in combination with -ddump-parsed-ast (page 256).	dynamic	
-dlinear-core-lint (page 265)	Turn on internal sanity checking	dynamic	
-dlint (page 265)	Enable several common internal sanity checkers	dynamic	
-dno-debug-output (page 264)	Suppress unsolicited debugging output	dynamic	-ddebug-output (page 264)
-dno-typeable-binds (page 267)	Don't generate bindings for Typeable methods	dynamic	
-dppr-case-as-let (page 264)	Print single alternative case expressions as strict lets.	dynamic	
-dppr-cols=⟨n⟩ (page 264)	Set the width of debugging output. For example `-dppr-cols200`	dynamic	
-dppr-debug (page 256)	Turn on debug printing (more verbose)	dynamic	
-dppr-user-length (page 264)	Set the depth for printing expressions in error msgs	dynamic	
-drule-check=⟨str⟩ (page 258)	Dump information about potential rule application	dynamic	
-dshow-passes (page 256)	Print out each pass name as it happens	dynamic	
-dstg-lint (page 265)	STG pass sanity checking	dynamic	
-dsuppress-all (page 264)	In dumps, suppress everything (except for uniques) that is suppressible.	dynamic	

Continued on next page

Table 25 – continued from previous page

Flag	Description	Type	Reverse
-dsuppress-coercion-types (page 265)	Suppress the printing of coercion types in Core dumps to make them shorter	dynamic	
-dsuppress-coercions (page 265)	Suppress the printing of coercions in Core dumps to make them shorter	dynamic	
-dsuppress-core-sizes (page 265)	Suppress the printing of core size stats per binding (since 9.4)	dynamic	
-dsuppress-idinfo (page 264)	Suppress extended information about identifiers where they are bound	dynamic	
-dsuppress-module-prefixes (page 264)	Suppress the printing of module qualification prefixes	dynamic	
-dsuppress-stg-free-vars (page 265)	Suppress the printing of closure free variable lists in STG output	dynamic	
-dsuppress-stg-reps (page 265)	Suppress rep annotations on STG args.	dynamic	
-dsuppress-ticks (page 264)	Suppress "ticks" in the pretty-printer output.	dynamic	
-dsuppress-timestamps (page 264)	Suppress timestamps in dumps	dynamic	
-dsuppress-type-applications (page 265)	Suppress type applications	dynamic	
-dsuppress-type-signatures (page 265)	Suppress type signatures	dynamic	
-dsuppress-unfoldings (page 264)	Suppress the printing of the stable unfolding of a variable at its binding site	dynamic	
-dsuppress-uniques (page 264)	Suppress the printing of uniques in debug output (easier to use `diff`)	dynamic	
-dsuppress-var-kinds (page 265)	Suppress the printing of variable kinds	dynamic	
-dtag-inference-checks (page 267)	Affirm tag inference results are correct at runtime.	dynamic	
-dth-dec-file (page 257)	Dump evaluated TH declarations into **.th.hs* files	dynamic	
-dunique-increment=⟨i⟩ (page 267)	Set the increment for the generated `Unique`'s to ⟨i⟩.	dynamic	
-dverbose-core2core (page 258)	Show output from each core-to-core pass	dynamic	
-dverbose-stg2stg (page 260)	Show output from each STG-to-STG pass	dynamic	
-falignment-sanitisation (page 266)	Compile with alignment checks for all info table dereferences.	dynamic	
-fcatch-nonexhaustive-cases (page 266)	Add a default `error` alternative to case expressions without a default alternative.	dynamic	
-fcheck-prim-bounds (page 266)	Instrument array primops with bounds checks.	dynamic	
-fcmm-thread-sanitizer (page 266)	Enable ThreadSanitizer instrumentation of memory accesses.	dynamic	
-fdistinct-constructor-tables (page 688)	Generate a fresh info table for each usage of a data constructor.	dynamic	

Continued on next page

Table 25 – continued from previous page

Flag	Description	Type	Reverse
-fdump-with-ways (page 255)	Include the tag of the enabled ways in the extension of dump files.	dynamic	
-finfo-table-map (page 687)	Embed a lookup table in the generated binary which maps the address of an info table to the source position the closure originated from.	dynamic	
-finfo-table-map-with-fallback (page 687)	Include info tables with no source location information in the info table map.	dynamic	*-fno-info-table-map* (page 687)
-finfo-table-map-with-stack (page 687)	Include info tables for `STACK` closures in the info table map.	dynamic	*-fno-info-table-map* (page 687)
-fllvm-fill-undef-with-garbage (page 266)	Instruct LLVM to fill dead STG registers with garbage	dynamic	
-fno-info-table-map-with-fallback (page 687)	Omit info tables with no source location information from the info table map.	dynamic	*-finfo-table-map-wi* (page 687)
-fno-info-table-map-with-stack (page 687)	Omit info tables for `STACK` closures from the info table map.	dynamic	*-finfo-table-map-wi* (page 687)
-forig-thunk-info (page 266)	Generate `stg_orig_thunk_info` stack frames on thunk entry	dynamic	
-fproc-alignment (page 266)	Align functions at given boundary.	dynamic	
-funoptimized-core-for-interpreter (page 267)	Disable optimizations with the interpreter	dynamic	*-fno-unoptimized-co* (page 267)
-g (page 681), *-g⟨n⟩* (page 681)	Produce DWARF debug information in compiled object files. ⟨n⟩ can be 0, 1, or 2, with higher numbers producing richer output. If ⟨n⟩ is omitted, level 2 is assumed.	dynamic	

5.6.27 Miscellaneous compiler options

Flag	Description	Type	Reverse
-ddump-mod-cycles (page 217)	Dump module cycles	dynamic	
-fdefer-out-of-scope-variables (page 415)	Convert variable out of scope variables errors into warnings. Implied by *-fdefer-type-errors* (page 414). See also *-Wdeferred-out-of-scope-variables* (page 88).	dynamic	*-fno-defer-out-of-s* (page 415)

Continued on next page

Table 26 – continued from previous page

Flag	Description	Type	Reverse
-fdefer-type-errors (page 414)	Turn type errors into warnings, *deferring the error until runtime* (page 414). Implies *-fdefer-typed-holes* (page 414) and *-fdefer-out-of-scope-variables* (page 415). See also *-Wdeferred-type-errors* (page 88).	dynamic	*-fno-defer-type-err* (page 414)
-fdefer-typed-holes (page 414)	Convert *typed hole* (page 304) errors into warnings, *deferring the error until runtime* (page 414). Implied by *-fdefer-type-errors* (page 414). See also *-Wtyped-holes* (page 88).	dynamic	*-fno-defer-typed-ho* (page 414)
-fexternal-interpreter (page 60)	Run interpreted code in a separate process	dynamic	
-ffamily-application-cache (page 347)	Use a cache when reducing type family applications	dynamic	*-fno-family-applica* (page 347)
-fglasgow-exts (page 273)	Deprecated. Enable most language extensions; see *Controlling editions and extensions* (page 269) for exactly which ones.	dynamic	*-fno-glasgow-exts* (page 273)
-fno-safe-haskell (page 587)	Disable *Safe Haskell* (page 577)	dynamic	
-ghcversion-file ⟨path to ghcversion.h⟩ (page 84)	(GHC as a C compiler only) Use this `ghcversion.h` file	dynamic	
-H ⟨size⟩ (page 84)	Set the minimum size of the heap to ⟨size⟩	dynamic	
-hidden-module ⟨module name⟩ (page 74)	A module which should not be visible outside its unit.	dynamic	
-j[⟨n⟩] (page 72)	When compiling with *--make* (page 68), compile ⟨n⟩ modules in parallel.	dynamic	
-jsem (page 72)	When compiling with *--make* (page 68), coordinate with other processes through the semaphore ⟨sem⟩ to compile modules in parallel.	dynamic	
-reexported-module ⟨module name⟩ (page 74)	A module which should be reexported from this unit.	dynamic	
-this-package-name ⟨unit-id⟩ (page 73)	The name of the package which this module would be part of when installed.	dynamic	
-unit @⟨filename⟩ (page 73)	Specify the options to build a specific unit.	dynamic	
-working-dir ⟨dir⟩ (page 73)	Specify the directory a unit is expected to be compiled in.	dynamic	

5.7 Runtime system (RTS) options

To make an executable program, the GHC system compiles your code and then links it with a non-trivial runtime system, which handles storage management, thread scheduling, profiling, and so on.

The RTS has a lot of options to control its behaviour. For example, you can change the context-switch interval, the default size of the heap, and enable heap profiling. These options can be passed to the runtime system in a variety of different ways; the next section (*Setting RTS options* (page 179)) describes the various methods, and the following sections describe the RTS options themselves.

5.7.1 Setting RTS options

There are four ways to set RTS options:

- on the command line between `+RTS ... -RTS`, when running the program (*Setting RTS options on the command line* (page 179))
- at compile-time, using *-with-rtsopts=⟨opts⟩* (page 249) (*Setting RTS options at compile time* (page 180))
- with the environment variable *GHCRTS* (page 180) (*Setting RTS options with the GHCRTS environment variable* (page 180))
- by overriding "hooks" in the runtime system (*"Hooks" to change RTS behaviour* (page 180))

5.7.1.1 Setting RTS options on the command line

If you set the *-rtsopts[=⟨none|some|all|ignore|ignoreAll⟩]* (page 248) flag appropriately when linking (see *Options affecting linking* (page 245)), you can give RTS options on the command line when running your program.

When your Haskell program starts up, the RTS extracts command-line arguments bracketed between `+RTS` and `-RTS` as its own. For example:

```
$ ghc prog.hs -rtsopts
[1 of 1] Compiling Main             ( prog.hs, prog.o )
Linking prog ...
$ ./prog -f +RTS -H32m -S -RTS -h foo bar
```

The RTS will snaffle `-H32m -S` for itself, and the remaining arguments `-f -h foo bar` will be available to your program if/when it calls `System.Environment.getArgs`.

No `-RTS` option is required if the runtime-system options extend to the end of the command line, as in this example:

```
% hls -ltr /usr/etc +RTS -A5m
```

If you absolutely positively want all the rest of the options in a command line to go to the program (and not the RTS), use a `--RTS` or `--`. The difference is that `--RTS` will not be passed to the program, while `--` will.

As always, for RTS options that take ⟨size⟩s: If the last character of ⟨size⟩ is a K or k, multiply by 1024; if an M or m, by 1024*1024; if a G or G, by 1024^3. (And any wraparound in the counters is *your* fault!)

Giving a `+RTS -?` RTS option will print out the RTS options actually available in your program (which vary, depending on how you compiled).

Note: Since GHC is itself compiled by GHC, you can change RTS options in the compiler using the normal `+RTS ... -RTS` combination. For instance, to set the maximum heap size for a compilation to 128M, you would add `+RTS -M128m -RTS` to the command line.

5.7.1.2 Setting RTS options at compile time

GHC lets you change the default RTS options for a program at compile time, using the `-with-rtsopts` flag (*Options affecting linking* (page 245)). A common use for this is to give your program a default heap and/or stack size that is greater than the default. For example, to set `-H128m -K64m`, link with `-with-rtsopts="-H128m -K64m"`.

5.7.1.3 Setting RTS options with the `GHCRTS` environment variable

`GHCRTS`

If the `-rtsopts` flag is set to something other than `none` or `ignoreAll` when linking, RTS options are also taken from the environment variable *`GHCRTS`* (page 180). For example, to set the maximum heap size to 2G for all GHC-compiled programs (using an `sh`-like shell):

```
GHCRTS='-M2G'
export GHCRTS
```

RTS options taken from the *`GHCRTS`* (page 180) environment variable can be overridden by options given on the command line.

Tip: Setting something like `GHCRTS=-M2G` in your environment is a handy way to avoid Haskell programs growing beyond the real memory in your machine, which is easy to do by accident and can cause the machine to slow to a crawl until the OS decides to kill the process (and you hope it kills the right one).

5.7.1.4 "Hooks" to change RTS behaviour

GHC lets you exercise rudimentary control over certain RTS settings for any given program, by compiling in a "hook" that is called by the run-time system. The RTS contains stub definitions for these hooks, but by writing your own version and linking it on the GHC command line, you can override the defaults.

Owing to the vagaries of DLL linking, these hooks don't work under Windows when the program is built dynamically.

Runtime events

You can change the messages printed when the runtime system "blows up," e.g., on stack overflow. The hooks for these are as follows:

void **OutOfHeapHook**(unsigned *long*, unsigned *long*)
: The heap-overflow message.

void **StackOverflowHook**(long *int*)
: The stack-overflow message.

void **MallocFailHook**(long *int*)
: The message printed if `malloc` fails.

Event log output

Furthermore GHC lets you specify the way event log data (see *-l ⟨flags⟩* (page 195)) is written through a custom *EventLogWriter* (page 181):

size_t

Hidden

EventLogWriter
: A sink of event-log data.

void **initEventLogWriter**(void)
: Initializes your *EventLogWriter* (page 181). This is optional.

bool **writeEventLog**(void **eventlog*, size_t *eventlog_size*)
: Hands buffered event log data to your event log writer. Return true on success. Required for a custom *EventLogWriter* (page 181).

Note that this function may be called by multiple threads simultaneously.

void **flushEventLog**(void)
: Flush buffers (if any) of your custom *EventLogWriter* (page 181). This can be `NULL`.

Note that this function may be called by multiple threads simultaneously.

void **stopEventLogWriter**(void)
: Called when event logging is about to stop. This can be `NULL`.

To use an *EventLogWriter* (page 181) the RTS API provides the following functions:

EventLogStatus (page 181) **eventLogStatus**(void)
: Query whether the current runtime system supports the eventlog (e.g. whether the current executable was linked with *-eventlog* (page 248)) and, if it is supported, whether it is currently logging.

bool **startEventLogging**(const *EventLogWriter* (page 181) **writer*)
: Start logging events to the given *EventLogWriter* (page 181). Returns true on success or false if another writer has already been configured.

void **endEventLogging**()
: Tear down the active *EventLogWriter* (page 181).

where the enum *EventLogStatus* (page 181) is:

EventLogStatus

- `EVENTLOG_NOT_SUPPORTED`: The runtime system wasn't compiled with eventlog support.
- `EVENTLOG_NOT_CONFIGURED`: An *EventLogWriter* (page 181) has not yet been configured.
- `EVENTLOG_RUNNING`: An *EventLogWriter* (page 181) has been configured and is running.

5.7.2 Miscellaneous RTS options

--install-signal-handlers=⟨yes|no⟩
If yes (the default), the RTS installs signal handlers to catch things like `Ctrl-C`. This option is primarily useful for when you are using the Haskell code as a DLL, and want to set your own signal handlers.

Note that even with `--install-signal-handlers=no`, the RTS interval timer signal is still enabled. The timer signal is either SIGVTALRM or SIGALRM, depending on the RTS configuration and OS capabilities. To disable the timer signal, use the `-V0` RTS option (see *-V ⟨secs⟩* (page 659)).

--install-seh-handlers=⟨yes|no⟩
If yes (the default), the RTS on Windows installs exception handlers to catch unhandled exceptions using the Windows exception handling mechanism. This option is primarily useful for when you are using the Haskell code as a DLL, and don't want the RTS to ungracefully terminate your application on errors such as segfaults.

--generate-crash-dumps
If yes (the default), the RTS on Windows will generate a core dump on any crash. These dumps can be inspected using debuggers such as WinDBG. The dumps record all code, registers and threading information at the time of the crash. Note that this implies `--install-seh-handlers=yes`.

--generate-stack-traces=<yes|no>
If yes (the default), the RTS on Windows will generate a stack trace on crashes if exception handling are enabled. In order to get more information in compiled executables, C code or DLLs symbols need to be available.

--disable-delayed-os-memory-return
If given, uses `MADV_DONTNEED` instead of `MADV_FREE` on platforms where this results in more accurate resident memory usage of the program as shown in memory usage reporting tools (e.g. the `RSS` column in `top` and `htop`).

Using this is expected to make the program slightly slower.

On Linux, MADV_FREE is newer and faster because it can avoid zeroing pages if they are re-used by the process later (see `man 2 madvise`), but for the trade-off that memory inspection tools like `top` will not immediately reflect the freeing in their display of resident memory (RSS column): Only under memory pressure will Linux actually remove the freed pages from the process and update its RSS statistics. Until then, the pages show up as `LazyFree` in `/proc/PID/smaps` (see `man 5 proc`).

The delayed RSS update can confuse programmers debugging memory issues, production memory monitoring tools, and end users who may complain about undue memory usage shown in reporting tools, so with this flag it can be turned off.

-xp
On 64-bit machines, the runtime linker usually needs to map object code into the low 2Gb

of the address space, due to the x86_64 small memory model where most symbol references are 32 bits. The problem is that this 2Gb of address space can fill up, especially if you're loading a very large number of object files into GHCi.

This flag offers a workaround, albeit a slightly convoluted one. To be able to load an object file outside of the low 2Gb, the object code needs to be compiled with `-fPIC -fexternal-dynamic-refs`. When the `+RTS -xp` flag is passed, the linker will assume that all object files were compiled with `-fPIC -fexternal-dynamic-refs` and load them anywhere in the address space. It's up to you to arrange that the object files you load (including all packages) were compiled in the right way. If this is not the case for an object, the linker will probably fail with an error message when the problem is detected.

On some platforms where PIC is always the case, e.g. macOS and OpenBSD on x86_64, and macOS and Linux on aarch64 this flag is enabled by default. One repercussion of this is that referenced system libraries also need to be compiled with `-fPIC` if we need to load them in the runtime linker.

-xm ⟨address⟩

> **Warning:** This option is for working around memory allocation problems only. Do not use unless GHCi fails with a message like "`failed to mmap() memory below 2Gb`". Consider recompiling the objects with `-fPIC -fexternal-dynamic-refs` and using the `-xp` flag instead. If you need to use this option to get GHCi working on your machine, please file a bug.

On 64-bit machines, the RTS needs to allocate memory in the low 2Gb of the address space. Support for this across different operating systems is patchy, and sometimes fails. This option is there to give the RTS a hint about where it should be able to allocate memory in the low 2Gb of the address space. For example, `+RTS -xm20000000 -RTS` would hint that the RTS should allocate starting at the 0.5Gb mark. The default is to use the OS's built-in support for allocating memory in the low 2Gb if available (e.g. `mmap` with `MAP_32BIT` on Linux), or otherwise `-xm40000000`.

-xq ⟨size⟩

Default 100k

This option relates to allocation limits; for more about this see GHC.Conc.enableAllocationLimit. When a thread hits its allocation limit, the RTS throws an exception to the thread, and the thread gets an additional quota of allocation before the exception is raised again, the idea being so that the thread can execute its exception handlers. The `-xq` controls the size of this additional quota.

-xr ⟨size⟩

Default 1T

This option controls the size of virtual memory address space reserved by the two step allocator on a 64-bit platform. It can be useful in scenarios where even reserving a large address range without committing can be expensive (e.g. WSL1), or when you actually have enough physical memory and want to support a Haskell heap larger than 1T. `-xr` is a no-op if GHC is configured with `--disable-large-address-space` or if the platform is 32-bit.

5.7.3 RTS options to control the garbage collector

There are several options to give you precise control over garbage collection. Hopefully, you won't need any of these in normal operation, but there are several things that can be tweaked for maximum performance.

--copying-gc

Default on

Since 8.10.2

Reverse –nonmoving-gc

Uses the generational copying garbage collector for all generations. This is the default.

--nonmoving-gc

Default off

Since 8.10.1

Reverse –copying-gc

Enable the concurrent mark-and-sweep garbage collector for old generation collectors. Typically GHC uses a stop-the-world copying garbage collector for all generations. This can cause long pauses in execution during major garbage collections. *--nonmoving-gc* (page 184) enables the use of a concurrent mark-and-sweep garbage collector for oldest generation collections. Under this collection strategy oldest-generation garbage collection can proceed concurrently with mutation.

Note that *--nonmoving-gc* (page 184) cannot be used with `-G1`, *profiling* (page 665) nor *-c* (page 186).

-xn

Default off

Since 8.10.1

An alias for *--nonmoving-gc* (page 184)

--nonmoving-dense-allocator-count=⟨count⟩

Default 16

Since 9.10.1

Reverse none

Specify the amount of dense allocators used by the non-moving garbage collector.

Increasing this value is likely to decrease the amount of memory lost to internal fragmentation while marginally increasing the baseline memory requirements and potentially regressing other metrics.

Large values are likely to lead to diminishing returns as , in practice, the Haskell heap tends to be dominated by small objects.

-w

Default off

Since a long time ago

Reverse none

Uses a mark-region garbage collection strategy for the oldest-generation heap. Note that this cannot be used in conjunction with heap profiling (*-hT* (page 194)) unless linked against the profiling runtime system with *-prof* (page 656).

-A ⟨size⟩

Default 4MB

Set the allocation area size used by the garbage collector. The allocation area (actually generation 0 step 0) is fixed and is never resized (unless you use *-H [⟨size⟩]* (page 188), below).

Optimal settings depend on the actual machine, program, and other RTS options. Increasing the allocation area size means worse cache behaviour but fewer garbage collections and less promotion.

In general settings >= 4MB can reduce performance in some cases, in particular for single threaded operation. However in a parallel setting increasing the allocation area to `16MB`, or even `64MB` can increase gc throughput significantly.

With only 1 generation (e.g. `-G1`, see *-G ⟨generations⟩* (page 187)) the `-A` option specifies the minimum allocation area, since the actual size of the allocation area will be resized according to the amount of data in the heap (see *-F ⟨factor⟩* (page 186), below).

When heap profiling using a smaller allocation area can increase accuracy as more frequent major garbage collections also results in more frequent heap snapshots

-AL ⟨size⟩

Default *-A* (page 185) value

Since 8.2.1

Sets the limit on the total size of "large objects" (objects larger than about 3KB) that can be allocated before a GC is triggered. By default this limit is the same as the *-A* (page 185) value.

Large objects are not allocated from the normal allocation area set by the `-A` flag, which is why there is a separate limit for these. Large objects tend to be much rarer than small objects, so most programs hit the `-A` limit before the `-AL` limit. However, the `-A` limit is per-capability, whereas the `-AL` limit is global, so as `-N` gets larger it becomes more likely that we hit the `-AL` limit first. To counteract this, it might be necessary to use a larger `-AL` limit when using a large `-N`.

To see whether you're making good use of all the memory reseverd for the allocation area (`-A` times `-N`), look at the output of `+RTS -S` and check whether the amount of memory allocated between GCs is equal to `-A` times `-N`. If not, there are two possible remedies: use `-n` to set a nursery chunk size, or use `-AL` to increase the limit for large objects.

-O ⟨size⟩

Default 1m

Set the minimum size of the old generation.

The old generation is collected whenever it grows to this size or the value of the *-F ⟨factor⟩* (page 186) option multiplied by the size of the live data at the previous major collection, whichever is larger.

-n ⟨size⟩

Default 4m with *-A16m* (page 185) or larger, otherwise 0.

Set the allocation area chunksize. Setting `-n0` means the allocation area is not divided into chunks.

[Example: `-n4m`] When set to a non-zero value, this option divides the allocation area (`-A` value) into chunks of the specified size. During execution, when a processor exhausts its current chunk, it is given another chunk from the pool until the pool is exhausted, at which point a collection is triggered.

This option is only useful when running in parallel (`-N2` or greater). It allows the processor cores to make better use of the available allocation area, even when cores are allocating at different rates. Without `-n`, each core gets a fixed-size allocation area specified by the `-A`, and the first core to exhaust its allocation area triggers a GC across all the cores. This can result in a collection happening when the allocation areas of some cores are only partially full, so the purpose of the `-n` is to allow cores that are allocating faster to get more of the allocation area. This means less frequent GC, leading a lower GC overhead for the same heap size.

This is particularly useful in conjunction with larger `-A` values, for example `-A64m -n4m` is a useful combination on larger core counts (8+).

-c

Use a compacting algorithm for collecting the oldest generation. By default, the oldest generation is collected using a copying algorithm; this option causes it to be compacted in-place instead. The compaction algorithm is slower than the copying algorithm, but the savings in memory use can be considerable.

For a given heap size (using the *-H [⟨size⟩]* (page 188) option), compaction can in fact reduce the GC cost by allowing fewer GCs to be performed. This is more likely when the ratio of live data to heap size is high, say greater than 30%.

Note: Compaction doesn't currently work when a single generation is requested using the `-G1` option.

-c ⟨n⟩

Default 30

Automatically enable compacting collection when the live data exceeds ⟨n⟩% of the maximum heap size (see the *-M ⟨size⟩* (page 190) option). Note that the maximum heap size is unlimited by default, so this option has no effect unless the maximum heap size is set with *-M ⟨size⟩* (page 190).

-F ⟨factor⟩

Default 2

This option controls the amount of memory reserved for the older generations (and in the case of a two space collector the size of the allocation area) as a factor of the amount of live data. For example, if there was 2M of live data in the oldest generation when we last collected it, then by default we'll wait until it grows to 4M before collecting it again.

The default seems to work well here. If you have plenty of memory, it is usually better to use `-H ⟨size⟩` (see *-H [⟨size⟩]* (page 188)) than to increase *-F ⟨factor⟩* (page 186).

The *-F ⟨factor⟩* (page 186) setting will be automatically reduced by the garbage collector when the maximum heap size (the *-M ⟨size⟩* (page 190) setting) is approaching.

-Fd ⟨factor⟩

Default 4

The inverse rate at which unused memory is returned to the OS when it is no longer needed. After a large amount of allocation the RTS will start by retaining a lot of allocated blocks in case it will need them again shortly but then it will gradually release them based on the *-Fd ⟨factor⟩* (page 186). On each subsequent major collection which is not caused by a heap overflow a little more memory will attempt to be returned until the amount retained is similar to the amount of live bytes.

Increasing this factor will make the rate memory is returned slower, decreasing it will make memory be returned more eagerly. Setting it to 0 will disable the memory return (which will emulate the behaviour in releases prior to 9.2).

-G ⟨generations⟩

Default 2

Set the number of generations used by the garbage collector. The default of 2 seems to be good, but the garbage collector can support any number of generations. Anything larger than about 4 is probably not a good idea unless your program runs for a *long* time, because the oldest generation will hardly ever get collected.

Specifying 1 generation with `+RTS -G1` gives you a simple 2-space collector, as you would expect. In a 2-space collector, the *-A ⟨size⟩* (page 185) option specifies the *minimum* allocation area size, since the allocation area will grow with the amount of live data in the heap. In a multi-generational collector the allocation area is a fixed size (unless you use the *-H [⟨size⟩]* (page 188) option).

-qg ⟨gen⟩

Default 0

Since 6.12.1

Use parallel GC in generation ⟨gen⟩ and higher. Omitting ⟨gen⟩ turns off the parallel GC completely, reverting to sequential GC.

The default parallel GC settings are usually suitable for parallel programs (i.e. those using GHC.Conc.par, Strategies, or with multiple threads). However, it is sometimes beneficial to enable the parallel GC for a single-threaded sequential program too, especially if the program has a large amount of heap data and GC is a significant fraction of runtime. To use the parallel GC in a sequential program, enable the parallel runtime with a suitable *-N ⟨x⟩* (page 132) option, and additionally it might be beneficial to restrict parallel GC to the old generation with `-qg1`.

-qb ⟨gen⟩

Default 1 for *-A* (page 185) < 32M, 0 otherwise

Since 6.12.1

Use load-balancing in the parallel GC in generation ⟨gen⟩ and higher. Omitting ⟨gen⟩ disables load-balancing entirely.

Load-balancing shares out the work of GC between the available cores. This is a good idea when the heap is large and we need to parallelise the GC work, however it is also pessimal for the short young-generation collections in a parallel program, because it can harm locality by moving data from the cache of the CPU where is it being used to the cache of another CPU. Hence the default is to do load-balancing only in the old-generation. In fact, for a parallel program it is sometimes beneficial to disable load-balancing entirely with `-qb`.

-qn ⟨x⟩

Default the value of *-N* (page 132) or the number of CPU cores, whichever is smaller.

Since 8.2.1

Set the number of threads to use for the parallel GC.

By default, all of the capabilities participate in parallel garbage collection. If we want to use a very large `-N` value, however, this can reduce the performance of the GC. For this reason, the `-qn` flag can be used to specify a lower number for the threads that should participate in GC. During GC, if there are more than this number of workers active, some of them will sleep for the duration of the GC.

The `-qn` flag may be useful when running with a large `-A` value (so that GC is infrequent), and a large `-N` value (so as to make use of hyperthreaded cores, for example). For example, on a 24-core machine with 2 hyperthreads per core, we might use `-N48 -qn24 -A128m` to specify that the mutator should use hyperthreads but the GC should only use real cores. Note that this configuration would use 6GB for the allocation area.

-H [⟨size⟩]

Default 0

This option provides a "suggested heap size" for the garbage collector. Think of `-Hsize` as a variable *-A ⟨size⟩* (page 185) option. It says: I want to use at least ⟨size⟩ bytes, so use whatever is left over to increase the `-A` value.

This option does not put a *limit* on the heap size: the heap may grow beyond the given size as usual.

If ⟨size⟩ is omitted, then the garbage collector will take the size of the heap at the previous GC as the ⟨size⟩. This has the effect of allowing for a larger `-A` value but without increasing the overall memory requirements of the program. It can be useful when the default small `-A` value is suboptimal, as it can be in programs that create large amounts of long-lived data.

-I ⟨seconds⟩

Default 0.3 seconds in the threaded runtime, 0 in the non-threaded runtime

Set the amount of idle time which must pass before a idle GC is performed. Setting `-I0` disables the idle GC.

In the threaded and SMP versions of the RTS (see *-threaded* (page 248), *Options affecting linking* (page 245)), a major GC is automatically performed if the runtime has been idle (no Haskell computation has been running) for a period of time.

For an interactive application, it is probably a good idea to use the idle GC, because this will allow finalizers to run and deadlocked threads to be detected in the idle time when no Haskell computation is happening. Also, it will mean that a GC is less likely to happen when the application is busy, and so responsiveness may be improved. However, if the amount of live data in the heap is particularly large, then the idle GC can cause a significant delay, and too small an interval could adversely affect interactive responsiveness.

The idle period timer only resets after some activity by a Haskell thread. If your program is doing literally nothing then after the first idle collection is triggered then no more future collections will be scheduled until more work is performed.

This is an experimental feature, please let us know if it causes problems and/or could benefit from further tuning.

-Iw ⟨seconds⟩

Default 0 seconds

Set the minimum wait time between runs of the idle GC.

By default, if idle GC is enabled in the threaded runtime, a major GC will be performed every time the process goes idle for a sufficiently long duration (see *-I ⟨seconds⟩* (page 188)). For large server processes accepting regular but infrequent requests (e.g., once per second), an expensive, major GC may run after every request. As an alternative to shutting off idle GC entirely (with `-I0`), a minimum wait time between idle GCs can be specified with this flag. For example, `-Iw60` will ensure that an idle GC runs at most once per minute.

This is an experimental feature, please let us know if it causes problems and/or could benefit from further tuning.

-ki ⟨size⟩

Default 1k

Set the initial stack size for new threads.

Thread stacks (including the main thread's stack) live on the heap. As the stack grows, new stack chunks are added as required; if the stack shrinks again, these extra stack chunks are reclaimed by the garbage collector. The default initial stack size is deliberately small, in order to keep the time and space overhead for thread creation to a minimum, and to make it practical to spawn threads for even tiny pieces of work.

Note: This flag used to be simply `-k`, but was renamed to `-ki` in GHC 7.2.1. The old name is still accepted for backwards compatibility, but that may be removed in a future version.

-kc ⟨size⟩

Default 32k

Set the size of "stack chunks". When a thread's current stack overflows, a new stack chunk is created and added to the thread's stack, until the limit set by *-K ⟨size⟩* (page 189) is reached.

The advantage of smaller stack chunks is that the garbage collector can avoid traversing stack chunks if they are known to be unmodified since the last collection, so reducing the chunk size means that the garbage collector can identify more stack as unmodified, and the GC overhead might be reduced. On the other hand, making stack chunks too small adds some overhead as there will be more overflow/underflow between chunks. The default setting of 32k appears to be a reasonable compromise in most cases.

-kb ⟨size⟩

Default 1k

Sets the stack chunk buffer size. When a stack chunk overflows and a new stack chunk is created, some of the data from the previous stack chunk is moved into the new chunk, to avoid an immediate underflow and repeated overflow/underflow at the boundary. The amount of stack moved is set by the `-kb` option.

Note that to avoid wasting space, this value should typically be less than 10% of the size of a stack chunk (*-kc ⟨size⟩* (page 189)), because in a chain of stack chunks, each chunk will have a gap of unused space of this size.

-K ⟨size⟩

Default 80% of physical memory

Set the maximum stack size for an individual thread to ⟨size⟩ bytes. If the thread attempts to exceed this limit, it will be sent the `StackOverflow` exception. The limit can be disabled entirely by specifying a size of zero.

This option is there mainly to stop the program eating up all the available memory in the machine if it gets into an infinite loop.

-m ⟨n⟩

Default 3%

Minimum % ⟨n⟩ of heap which must be available for allocation.

-M ⟨size⟩

Default unlimited

Set the maximum heap size to ⟨size⟩ bytes. The heap normally grows and shrinks according to the memory requirements of the program. The only reason for having this option is to stop the heap growing without bound and filling up all the available swap space, which at the least will result in the program being summarily killed by the operating system.

The maximum heap size also affects other garbage collection parameters: when the amount of live data in the heap exceeds a certain fraction of the maximum heap size, compacting collection will be automatically enabled for the oldest generation, and the `-F` parameter will be reduced in order to avoid exceeding the maximum heap size.

-Mgrace=⟨size⟩

Default 1M

If the program’s heap exceeds the value set by *-M ⟨size⟩* (page 190), the RTS throws an exception to the program, and the program gets an additional quota of allocation before the exception is raised again, the idea being so that the program can execute its exception handlers. `-Mgrace=` controls the size of this additional quota.

--numa

--numa=<mask>

Enable NUMA-aware memory allocation in the runtime (only available with `-threaded`, and only on Linux and Windows currently).

Background: some systems have a Non-Uniform Memory Architecture, whereby main memory is split into banks which are “local” to specific CPU cores. Accessing local memory is faster than accessing remote memory. The OS provides APIs for allocating local memory and binding threads to particular CPU cores, so that we can ensure certain memory accesses are using local memory.

The `--numa` option tells the RTS to tune its memory usage to maximize local memory accesses. In particular, the RTS will:

- Determine the number of NUMA nodes (N) by querying the OS.
- Manage separate memory pools for each node.
- Map capabilities to NUMA nodes. Capability C is mapped to NUMA node C mod N.
- Bind worker threads on a capability to the appropriate node.
- Allocate the nursery from node-local memory.
- Perform other memory allocation, including in the GC, from node-local memory.

- When load-balancing, we prefer to migrate threads to another Capability on the same node.

The `--numa` flag is typically beneficial when a program is using all cores of a large multi-core NUMA system, with a large allocation area (`-A`). All memory accesses to the allocation area will go to local memory, which can save a significant amount of remote memory access. A runtime speedup on the order of 10% is typical, but can vary a lot depending on the hardware and the memory behaviour of the program.

Note that the RTS will not set CPU affinity for bound threads and threads entering Haskell from C/C++, so if your program uses bound threads you should ensure that each bound thread calls the RTS API *rts_setInCallCapability(c,1)* from C/C++ before calling into Haskell. Otherwise there could be a mismatch between the CPU that the thread is running on and the memory it is using while running Haskell code, which will negate any benefits of `--numa`.

If given an explicit <mask>, the <mask> is interpreted as a bitmap that indicates the NUMA nodes on which to run the program. For example, `--numa=3` would run the program on NUMA nodes 0 and 1.

`--long-gc-sync`
`--long-gc-sync``=<seconds>`

When a GC starts, all the running mutator threads have to stop and synchronise. The period between when the GC is initiated and all the mutator threads are stopped is called the GC synchronisation phase. If this phase is taking a long time (longer than 1ms is considered long), then it can have a severe impact on overall throughput.

A long GC sync can be caused by a mutator thread that is inside an `unsafe` FFI call, or running in a loop that doesn't allocate memory and so doesn't yield. To fix the former, make the call `safe`, and to fix the latter, either avoid calling the code in question or compile it with *`-fomit-yields`* (page 121).

By default, the flag will cause a warning to be emitted to stderr when the sync time exceeds the specified time. This behaviour can be overridden, however: the `longGCSync()` hook is called when the sync time is exceeded during the sync period, and the `longGCSyncEnd()` hook at the end. Both of these hooks can be overridden in the `RtsConfig` when the runtime is started with `hs_init_ghc()`. The default implementations of these hooks (`LongGcSync()` and `LongGCSyncEnd()` respectively) print warnings to stderr.

One way to use this flag is to set a breakpoint on `LongGCSync()` in the debugger, and find the thread that is delaying the sync. You probably want to use *`-g`* (page 681) to provide more info to the debugger.

The GC sync time, along with other GC stats, are available by calling the `getRTSStats()` function from C, or `GHC.Stats.getRTSStats` from Haskell.

5.7.4 RTS options to produce runtime statistics

`-T`
`-t` `[⟨file⟩]`
`-s` `[⟨file⟩]`
`-S` `[⟨file⟩]`
`--machine-readable`
`--internal-counters`

These options produce runtime-system statistics, such as the amount of time spent executing the program and in the garbage collector, the amount of memory allocated, the maximum size of the heap, and so on. The three variants give different levels of detail: -T collects the data but produces no output -t produces a single line of output in the same format as GHC's -Rghc-timing option, -s produces a more detailed summary at the end of the program, and -S additionally produces information about each and every garbage collection. Passing --internal-counters to a threaded runtime will cause a detailed summary to include various internal counts accumulated during the run; note that these are unspecified and may change between releases.

The output is placed in ⟨file⟩. If ⟨file⟩ is omitted, then the output is sent to stderr.

If you use the -T flag then, you should access the statistics using GHC.Stats.

If you use the -t flag then, when your program finishes, you will see something like this:

```
<<ghc: 36169392 bytes, 69 GCs, 603392/1065272 avg/max bytes residency (2 samples),
↪ 3M in use, 0.00 INIT (0.00 elapsed), 0.02 MUT (0.02 elapsed), 0.07 GC (0.07␣
↪elapsed) :ghc>>
```

This tells you:

- The total number of bytes allocated by the program over the whole run.
- The total number of garbage collections performed.
- The average and maximum "residency", which is the amount of live data in bytes. The runtime can only determine the amount of live data during a major GC, which is why the number of samples corresponds to the number of major GCs (and is usually relatively small). To get a better picture of the heap profile of your program, use the *-hT* (page 194) RTS option (*RTS options for profiling* (page 194)).
- The peak memory the RTS has allocated from the OS.
- The amount of CPU time and elapsed wall clock time while initialising the runtime system (INIT), running the program itself (MUT, the mutator), and garbage collecting (GC).

You can also get this in a more future-proof, machine readable format, with -t --machine-readable:

```
[("bytes allocated", "36169392")
,("num_GCs", "69")
,("average_bytes_used", "603392")
,("max_bytes_used", "1065272")
,("num_byte_usage_samples", "2")
,("peak_megabytes_allocated", "3")
,("init_cpu_seconds", "0.00")
,("init_wall_seconds", "0.00")
,("mutator_cpu_seconds", "0.02")
,("mutator_wall_seconds", "0.02")
,("GC_cpu_seconds", "0.07")
,("GC_wall_seconds", "0.07")
]
```

If you use the -s flag then, when your program finishes, you will see something like this (the exact details will vary depending on what sort of RTS you have, e.g. you will only see profiling data if your RTS is compiled for profiling):

```
    36,169,392 bytes allocated in the heap
     4,057,632 bytes copied during GC
     1,065,272 bytes maximum residency (2 sample(s))
```

(continues on next page)

(continued from previous page)

```
        54,312 bytes maximum slop
             3 MB total memory in use (0 MB lost due to fragmentation)

Generation 0:    67 collections,     0 parallel,  0.04s,  0.03s elapsed
Generation 1:     2 collections,     0 parallel,  0.03s,  0.04s elapsed

SPARKS: 359207 (557 converted, 149591 pruned)

INIT  time    0.00s  (  0.00s elapsed)
MUT   time    0.01s  (  0.02s elapsed)
GC    time    0.07s  (  0.07s elapsed)
EXIT  time    0.00s  (  0.00s elapsed)
Total time    0.08s  (  0.09s elapsed)

%GC time      89.5%  (75.3% elapsed)

Alloc rate    4,520,608,923 bytes per MUT second

Productivity  10.5% of total user, 9.1% of total elapsed
```

- The "bytes allocated in the heap" is the total bytes allocated by the program over the whole run.
- GHC uses a copying garbage collector by default. "bytes copied during GC" tells you how many bytes it had to copy during garbage collection.
- The maximum space actually used by your program is the "bytes maximum residency" figure. This is only checked during major garbage collections, so it is only an approximation; the number of samples tells you how many times it is checked.
- The "bytes maximum slop" tells you the most space that is ever wasted due to the way GHC allocates memory in blocks. Slop is memory at the end of a block that was wasted. There's no way to control this; we just like to see how much memory is being lost this way.
- The "total memory in use" tells you the peak memory the RTS has allocated from the OS.
- Next there is information about the garbage collections done. For each generation it says how many garbage collections were done, how many of those collections were done in parallel, the total CPU time used for garbage collecting that generation, and the total wall clock time elapsed while garbage collecting that generation.
- The `SPARKS` statistic refers to the use of `Control.Parallel.par` and related functionality in the program. Each spark represents a call to `par`; a spark is "converted" when it is executed in parallel; and a spark is "pruned" when it is found to be already evaluated and is discarded from the pool by the garbage collector. Any remaining sparks are discarded at the end of execution, so "converted" plus "pruned" does not necessarily add up to the total.
- Next there is the CPU time and wall clock time elapsed broken down by what the runtime system was doing at the time. INIT is the runtime system initialisation. MUT is the mutator time, i.e. the time spent actually running your code. GC is the time spent doing garbage collection. RP is the time spent doing retainer profiling. PROF is the time spent doing other profiling. EXIT is the runtime system shutdown time. And finally, Total is, of course, the total.

%GC time tells you what percentage GC is of Total. "Alloc rate" tells you the "bytes allocated in the heap" divided by the MUT CPU time. "Productivity" tells you what percentage of the Total CPU and wall clock elapsed times are spent in the mutator (MUT).

The `-S` flag, as well as giving the same output as the `-s` flag, prints information about each GC as it happens:

```
    Alloc    Copied     Live     GC     GC      TOT      TOT  Page Flts
    bytes     bytes     bytes   user   elap     user     elap
   528496     47728    141512  0.01   0.02     0.02     0.02    0    0  (Gen:  1)
[...]
   524944    175944   1726384  0.00   0.00     0.08     0.11    0    0  (Gen:  0)
```

For each garbage collection, we print:

- How many bytes we allocated this garbage collection.
- How many bytes we copied this garbage collection.
- How many bytes are currently live.
- How long this garbage collection took (CPU time and elapsed wall clock time).
- How long the program has been running (CPU time and elapsed wall clock time).
- How many page faults occurred this garbage collection.
- How many page faults occurred since the end of the last garbage collection.
- Which generation is being garbage collected.

5.7.5 RTS options for concurrency and parallelism

The RTS options related to concurrency are described in *Using Concurrent Haskell* (page 131), and those for parallelism in *RTS options for SMP parallelism* (page 132).

5.7.6 RTS options for profiling

Most profiling runtime options are only available when you compile your program for profiling (see *Compiler options for profiling* (page 656), and *RTS options for heap profiling* (page 665) for the runtime options). However, there is one profiling option that is available for ordinary non-profiled executables:

-hT
-h

Generates a basic heap profile, in the file `prog.hp`. To produce the heap profile graph, use **hp2ps** (see *hp2ps – Rendering heap profiles to PostScript* (page 670)). The basic heap profile is broken down by data constructor, with other types of closures (functions, thunks, etc.) grouped into broad categories (e.g. `FUN`, `THUNK`). To get a more detailed profile, use the full profiling support (*Profiling* (page 651)). Can be shortened to *-h* (page 194).

Note: The meaning of the shortened *-h* (page 194) is dependent on whether your program was compiled for profiling. (See *RTS options for heap profiling* (page 665) for details.)

-L ⟨n⟩

Default 25 characters

Sets the maximum length of the cost-centre names listed in the heap profile.

5.7.7 Tracing

When the program is linked with the *-eventlog* (page 248) option (*Options affecting linking* (page 245)), runtime events can be logged in several ways:

- In binary format to a file for later analysis by a variety of tools. One such tool is ThreadScope, which interprets the event log to produce a visual parallel execution profile of the program.
- In binary format to customized event log writer. This enables live analysis of the events while the program is running.
- As text to standard output, for debugging purposes.

-l ⟨flags⟩

Log events in binary format. Without any ⟨flags⟩ specified, this logs a default set of events, suitable for use with tools like ThreadScope.

Per default the events are written to *program*.eventlog though the mechanism for writing event log data can be overridden with a custom *EventLogWriter*.

For some special use cases you may want more control over which events are included. The ⟨flags⟩ is a sequence of zero or more characters indicating which classes of events to log. Currently these the classes of events that can be enabled/disabled:

- s — scheduler events, including Haskell thread creation and start/stop events. Enabled by default.
- g — GC events, including GC start/stop. Enabled by default.
- n — non-moving garbage collector (see *--nonmoving-gc* (page 184)) events including start and end of the concurrent mark and census information to characterise heap fragmentation. Disabled by default.
- p — parallel sparks (sampled). Enabled by default.
- f — parallel sparks (fully accurate). Disabled by default.
- T — *ticky-ticky profiler* (page 677) events (see *Ticky counters* (page 759) for details). Disabled by default.
- u — user events. These are events emitted from Haskell code using functions such as Debug.Trace.traceEvent. Enabled by default.

You can disable specific classes, or enable/disable all classes at once:

- a — enable all event classes listed above
- -⟨x⟩ — disable the given class of events, for any event class listed above
- -a — disable all classes

For example, -l-ag would disable all event classes (-a) except for GC events (g).

For spark events there are two modes: sampled and fully accurate. There are various events in the life cycle of each spark, usually just creating and running, but there are some more exceptional possibilities. In the sampled mode the number of occurrences of each kind of spark event is sampled at frequent intervals. In the fully accurate mode every spark event is logged individually. The latter has a higher runtime overhead and is not enabled by default.

The format of the log file is described in this users guide in *Eventlog encodings* (page 743) It can be parsed in Haskell using the ghc-events library. To dump the contents of a `.eventlog` file as text, use the tool `ghc-events show` that comes with the ghc-events package.

Each event is associated with a timestamp which is the number of nanoseconds since the start of executation of the running program. This is the elapsed time, not the CPU time.

-ol⟨filename⟩

Default ⟨program⟩.eventlog

Since 8.8

Sets the destination for the eventlog produced with the *-l ⟨flags⟩* (page 195) flag.

--eventlog-flush-interval=⟨seconds⟩

Default disabled

Since 9.2

When enabled, the eventlog will be flushed periodically every ⟨seconds⟩. This can be useful in live-monitoring situations where the eventlog is consumed in real-time by another process.

-v [⟨flags⟩]

Log events as text to standard output, instead of to the `.eventlog` file. The ⟨flags⟩ are the same as for `-l`, with the additional option `t` which indicates that the each event printed should be preceded by a timestamp value (in the binary `.eventlog` file, all events are automatically associated with a timestamp).

The debugging options `-Dx` also generate events which are logged using the tracing framework. By default those events are dumped as text to stdout (`-Dx` implies `-v`), but they may instead be stored in the binary eventlog file by using the `-l` option.

5.7.8 RTS options for Haskell program coverage

When a program is compiled with the *-fhpc* (page 674) flag, then the generated code is instrumented with instructions which keep track of which code was executed while the program runs. This functionality is implemented in the runtime system and can be controlled by the following flags.

--write-tix-file

Default enabled

Since 9.10

By default, the runtime system writes a file <program>.tix at the end of execution if the executable is compiled with the `-fhpc` option. This file is not written if the `--write-tix-file=no` option is passed to the runtime system.

This option is useful if you want to use the functionality provided by the `Trace.Hpc.Reflect` module of the hpc library. These functions allow to inspect the state of the Tix data structures during runtime, so that the executable can write Tix files to disk itself.

5.7.9 RTS options for hackers, debuggers, and over-interested souls

These RTS options might be used (a) to avoid a GHC bug, (b) to see "what's really happening", or (c) because you feel like it. Not recommended for everyday use!

-B

Sound the bell at the start of each garbage collection.

Oddly enough, people really do use this option! Our pal in Durham (England), Paul Callaghan, writes: "Some people here use it for a variety of purposes—honestly!—e.g., confirmation that the code/machine is doing something, infinite loop detection, gauging cost of recently added code. Certain people can even tell what stage [the program] is in by the beep pattern. But the major use is for annoying others in the same office…"

-D ⟨x⟩

An RTS debugging flag; only available if the program was linked with the *-debug* (page 248) option. Various values of ⟨x⟩ are provided to enable debug messages and additional runtime sanity checks in different subsystems in the RTS, for example `+RTS -Ds -RTS` enables debug messages from the scheduler. Use `+RTS -?` to find out which debug flags are supported.

Full list of currently supported flags:

-Ds `DEBUG: scheduler`

-Di `DEBUG: interpreter`

-Dw `DEBUG: weak`

-DG `DEBUG: gccafs`

-Dg `DEBUG: gc`

-Db `DEBUG: block`

-DS `DEBUG: sanity`

-DZ `DEBUG: zero freed memory on GC`

-Dt `DEBUG: stable`

-Dp `DEBUG: prof`

-Da `DEBUG: apply`

-Dl `DEBUG: linker`

-DL ``DEBUG: linker (verbose); implies :rts-flag:`-Dl` ``

-Dm `DEBUG: stm`

-Dn `DEBUG: non-moving garbage collector`

-Dz `DEBUG: stack squeezing`

-Dc `DEBUG: program coverage`

-Dr `DEBUG: sparks`

-DC `DEBUG: compact`

-Dk `DEBUG: continuation`

Debug messages will be sent to the binary event log file instead of stdout if the *-l ⟨flags⟩* (page 195) option is added. This might be useful for reducing the overhead of debug tracing.

To figure out what exactly they do, the least bad way is to grep the rts/ directory in the ghc code for macros like `DEBUG(scheduler` or `DEBUG_scheduler`.

-r ⟨file⟩

Produce "ticky-ticky" statistics at the end of the program run (only available if the program was linked with *-debug* (page 248)). The ⟨file⟩ business works just like on the *-S [⟨file⟩]* (page 191) RTS option, above.

For more information on ticky-ticky profiling, see *Using "ticky-ticky" profiling (for implementors)* (page 677).

-xc

(Only available when the program is compiled for profiling.) When an exception is raised in the program, this option causes a stack trace to be dumped to `stderr`.

This can be particularly useful for debugging: if your program is complaining about a `head []` error and you haven't got a clue which bit of code is causing it, compiling with `-prof -fprof-auto` (see *-prof* (page 656)) and running with `+RTS -xc -RTS` will tell you exactly the call stack at the point the error was raised.

The output contains one report for each exception raised in the program (the program might raise and catch several exceptions during its execution), where each report looks something like this:

```
*** Exception raised (reporting due to +RTS -xc), stack trace:
  GHC.List.CAF
  --> evaluated by: Main.polynomial.table_search,
  called from Main.polynomial.theta_index,
  called from Main.polynomial,
  called from Main.zonal_pressure,
  called from Main.make_pressure.p,
  called from Main.make_pressure,
  called from Main.compute_initial_state.p,
  called from Main.compute_initial_state,
  called from Main.CAF
  ...
```

The stack trace may often begin with something uninformative like `GHC.List.CAF`; this is an artifact of GHC's optimiser, which lifts out exceptions to the top-level where the profiling system assigns them to the cost centre "CAF". However, `+RTS -xc` doesn't just print the current stack, it looks deeper and reports the stack at the time the CAF was evaluated, and it may report further stacks until a non-CAF stack is found. In the example above, the next stack (after `--> evaluated by`) contains plenty of information about what the program was doing when it evaluated `head []`.

Implementation details aside, the function names in the stack should hopefully give you enough clues to track down the bug.

See also the function `traceStack` in the module `Debug.Trace` for another way to view call stacks.

-Z

Turn *off* update frame squeezing on context switch. (There's no particularly good reason to turn it off, except to ensure the accuracy of certain data collected regarding thunk entry counts.)

5.7.10 Getting information about the RTS

--info

It is possible to ask the RTS to give some information about itself. To do this, use the *--info* (page 199) flag, e.g.

```
$ ./a.out +RTS --info
[("GHC RTS", "YES")
,("GHC version", "6.7")
,("RTS way", "rts_p")
,("Host platform", "x86_64-unknown-linux")
,("Host architecture", "x86_64")
,("Host OS", "linux")
,("Host vendor", "unknown")
,("Build platform", "x86_64-unknown-linux")
,("Build architecture", "x86_64")
,("Build OS", "linux")
,("Build vendor", "unknown")
,("Target platform", "x86_64-unknown-linux")
,("Target architecture", "x86_64")
,("Target OS", "linux")
,("Target vendor", "unknown")
,("Word size", "64")
,("Compiler unregisterised", "NO")
,("Tables next to code", "YES")
,("Flag -with-rtsopts", "")
]
```

The information is formatted such that it can be read as a of type `[(String, String)]`. Currently the following fields are present:

GHC RTS Is this program linked against the GHC RTS? (always "YES").

GHC version The version of GHC used to compile this program.

RTS way The variant ("way") of the runtime. The most common values are `rts_v` (vanilla), `rts_thr` (threaded runtime, i.e. linked using the *-threaded* (page 248) option) and `rts_p` (profiling runtime, i.e. linked using the *-prof* (page 656) option). Other variants include `debug` (linked using *-debug* (page 248)), and `dyn` (the RTS is linked in dynamically, i.e. a shared library, rather than statically linked into the executable itself). These can be combined, e.g. you might have `rts_thr_debug_p`.

Target platformTarget architectureTarget OSTarget vendor These are the platform the program is compiled to run on.

Build platformBuild architectureBuild OSBuild vendor These are the platform where the program was built on. (That is, the target platform of GHC itself.) Ordinarily this is identical to the target platform. (It could potentially be different if cross-compiling.)

Host platformHost architectureHost OSHost vendor These are the platform where GHC itself was compiled. Again, this would normally be identical to the build and target platforms.

Word size Either `"32"` or `"64"`, reflecting the word size of the target platform.

Compiler unregistered Was this program compiled with an *"unregistered"* (page 237) version of GHC? (I.e., a version of GHC that has no platform-specific optimisations compiled in, usually because this is a currently unsupported platform.) This value will usually be no, unless you're using an experimental build of GHC.

Tables next to code Putting info tables directly next to entry code is a useful performance optimisation that is not available on all platforms. This field tells you whether the program has been compiled with this optimisation. (Usually yes, except on unusual platforms.)

Flag -with-rtsopts The value of the GHC flag *-with-rtsopts=⟨opts⟩* (page 249) at compile/link time.

5.8 Filenames and separate compilation

This section describes what files GHC expects to find, what files it creates, where these files are stored, and what options affect this behaviour.

Pathname conventions vary from system to system. In particular, the directory separator is "/" on Unix systems and "\" on Windows systems. In the sections that follow, we shall consistently use "/" as the directory separator; substitute this for the appropriate character for your system.

5.8.1 Haskell source files

Each Haskell source module should be placed in a file on its own.

Usually, the file should be named after the module name, replacing dots in the module name by directory separators. For example, on a Unix system, the module `A.B.C` should be placed in the file `A/B/C.hs`, relative to some base directory. If the module is not going to be imported by another module (`Main`, for example), then you are free to use any filename for it.

GHC assumes that source files are ASCII or UTF-8 only, other encoding are not recognised. However, invalid UTF-8 sequences will be ignored in comments, so it is possible to use other encodings such as Latin-1, as long as the non-comment source code is ASCII only.

5.8.2 Output files

When asked to compile a source file, GHC normally generates two files: an object file, and an interface file.

The object file, which normally ends in a `.o` suffix, contains the compiled code for the module.

The interface file, which normally ends in a `.hi` suffix, contains the information that GHC needs in order to compile further modules that depend on this module. It contains things like the types of exported functions, definitions of data types, and so on. It is stored in a binary format, so don't try to read one; use the *--show-iface ⟨file⟩* (page 69) option instead (see *Other options related to interface files* (page 205)).

You should think of the object file and the interface file as a pair, since the interface file is in a sense a compiler-readable description of the contents of the object file. If the interface file and object file get out of sync for any reason, then the compiler may end up making assumptions about the object file that aren't true; trouble will almost certainly follow. For this reason, we recommend keeping object files and interface files in the same place (GHC does this by default, but it is possible to override the defaults as we'll explain shortly).

Every module has a *module name* defined in its source code (`module A.B.C where ...`).

The name of the object file generated by GHC is derived according to the following rules, where ⟨osuf⟩ is the object-file suffix (this can be changed with the `-osuf` option).

- If there is no `-odir` option (the default), then the object filename is derived from the source filename (ignoring the module name) by replacing the suffix with ⟨osuf⟩.
- If `-odir ⟨dir⟩` has been specified, then the object filename is ⟨dir⟩/⟨mod⟩.⟨osuf⟩, where ⟨mod⟩ is the module name with dots replaced by slashes. GHC will silently create the necessary directory structure underneath ⟨dir⟩, if it does not already exist.

The name of the interface file is derived using the same rules, except that the suffix is ⟨hisuf⟩ (`.hi` by default) instead of ⟨osuf⟩, and the relevant options are *`-hidir ⟨dir⟩`* (page 203) and *`-hisuf ⟨suffix⟩`* (page 203) instead of *`-odir ⟨dir⟩`* (page 202) and *`-osuf ⟨suffix⟩`* (page 203) respectively.

For example, if GHC compiles the module `A.B.C` in the file `src/A/B/C.hs`, with no `-odir` or `-hidir` flags, the interface file will be put in `src/A/B/C.hi` and the object file in `src/A/B/C.o`.

For any module that is imported, GHC requires that the name of the module in the import statement exactly matches the name of the module in the interface file (or source file) found using the strategy specified in *The search path* (page 201). This means that for most modules, the source file name should match the module name.

However, note that it is reasonable to have a module `Main` in a file named `foo.hs`, but this only works because GHC never needs to search for the interface for module `Main` (because it is never imported). It is therefore possible to have several `Main` modules in separate source files in the same directory, and GHC will not get confused.

In batch compilation mode, the name of the object file can also be overridden using the *`-o ⟨file⟩`* (page 202) option, and the name of the interface file can be specified directly using the *`-ohi ⟨file⟩`* (page 203) option.

5.8.3 The search path

In your program, you import a module `Foo` by saying `import Foo`. In *`--make`* (page 68) mode or GHCi, GHC will look for a source file for `Foo` and arrange to compile it first. Without *`--make`* (page 68), GHC will look for the interface file for `Foo`, which should have been created by an earlier compilation of `Foo`.

The strategy for looking for source files is as follows: GHC keeps a list of directories called the search path. For each of these directories, it tries appending `⟨basename⟩.⟨extension⟩` to the directory, and checks whether the file exists. The value of ⟨basename⟩ is the module name with dots replaced by the directory separator ("/" or "\\", depending on the system), and ⟨extension⟩ is a source extension (`hs`, `lhs`) if we are in *`--make`* (page 68) mode or GHCi.

When looking for interface files in *`-c`* (page 68) mode, we look for interface files in the `-hidir`, if it's set. Otherwise the same strategy as for source files is used to try to locate the interface file.

For example, suppose the search path contains directories `d1`, `d2`, and `d3`, and we are in *`--make`* (page 68) mode looking for the source file for a module `A.B.C`. GHC will look in `d1/A/B/C.hs`, `d1/A/B/C.lhs`, `d2/A/B/C.hs`, and so on.

The search path by default contains a single directory: "`.`" (i.e. the current directory). The following options can be used to add to or change the contents of the search path:

`-i⟨dir⟩``[:⟨dir⟩]*`
: This flag appends a colon-separated list of `dirs` to the search path.

`-i`
: resets the search path back to nothing.

This isn't the whole story: GHC also looks for modules in pre-compiled libraries, known as packages. See the section on packages (*Packages* (page 219)) for details.

5.8.4 Redirecting the compilation output(s)

-o ⟨file⟩
GHC's compiled output normally goes into a `.hc`, `.o`, etc., file, depending on the last-run compilation phase. The option `-o file` re-directs the output of that last-run phase to ⟨file⟩.

Note: This "feature" can be counterintuitive: `ghc -C -o foo.o foo.hs` will put the intermediate C code in the file `foo.o`, name notwithstanding!

This option is most often used when creating an executable file, to set the filename of the executable. For example:

```
ghc -o prog --make Main
```

will compile the program starting with module `Main` and put the executable in the file `prog`.

Note: on Windows, if the result is an executable file, the extension "`.exe`" is added if the specified filename does not already have an extension. Thus

```
ghc -o foo Main.hs
```

will compile and link the module `Main.hs`, and put the resulting executable in `foo.exe` (not `foo`).

If you use `ghc --make` and you don't use the `-o`, the name GHC will choose for the executable will be based on the name of the file containing the module `Main`. Note that with GHC the `Main` module doesn't have to be put in file `Main.hs`. Thus both

```
ghc --make Prog
```

and

```
ghc --make Prog.hs
```

will produce `Prog` (or `Prog.exe` if you are on Windows).

-dyno ⟨file⟩
When using `-dynamic-too`, option `-dyno` ⟨suffix⟩ is the counterpart of `-o`. It redirects the dynamic output to ⟨file⟩.

-odir ⟨dir⟩
Redirects object files to directory ⟨dir⟩. For example:

```
$ ghc -c parse/Foo.hs parse/Bar.hs gurgle/Bumble.hs -odir `uname -m`
```

The object files, `Foo.o`, `Bar.o`, and `Bumble.o` would be put into a subdirectory named after the architecture of the executing machine (`x86`, `mips`, etc).

Note that the `-odir` option does *not* affect where the interface files are put; use the `-hidir` option for that. In the above example, they would still be put in `parse/Foo.hi`, `parse/Bar.hi`, and `gurgle/Bumble.hi`.

Please also note that when doing incremental compilation, this directory is where GHC looks into to find object files from previous builds.

-ohi ⟨file⟩
The interface output may be directed to another file `bar2/Wurble.iface` with the option `-ohi bar2/Wurble.iface` (not recommended).

> **Warning:** If you redirect the interface file somewhere that GHC can't find it, then the recompilation checker may get confused (at the least, you won't get any recompilation avoidance). We recommend using a combination of `-hidir` and `-hisuf` options instead, if possible.

To avoid generating an interface at all, you could use this option to redirect the interface into the bit bucket: `-ohi /dev/null`, for example.

-dynohi ⟨file⟩
When using `-dynamic-too`, option `-dynohi` ⟨file⟩ is the counterpart of `-ohi`. It redirects the dynamic interface output to ⟨file⟩.

-hidir ⟨dir⟩
Redirects all generated interface files into ⟨dir⟩, instead of the default.

Please also note that when doing incremental compilation (by `ghc --make` or `ghc -c`), this directory is where GHC looks into to find interface files.

-hiedir ⟨dir⟩
Redirects all generated extended interface files into ⟨dir⟩, instead of the default.

Please also note that when doing incremental compilation (by `ghc --make` or `ghc -c`), this directory is where GHC looks into to find extended interface files.

-stubdir ⟨dir⟩
Redirects all generated FFI stub files into ⟨dir⟩. Stub files are generated when the Haskell source contains a `foreign export` or `foreign import "&wrapper"` declaration (see *Using foreign export and foreign import ccall "wrapper" with GHC* (page 568)). The `-stubdir` option behaves in exactly the same way as `-odir` and `-hidir` with respect to hierarchical modules.

-dumpdir ⟨dir⟩
Redirects all dump files into ⟨dir⟩. Dump files are generated when `-ddump-to-file` is used with other `-ddump-*` flags.

-outputdir ⟨dir⟩
The `-outputdir` option is shorthand for the combination of *-odir ⟨dir⟩* (page 202), *-hidir ⟨dir⟩* (page 203), *-hiedir ⟨dir⟩* (page 203), *-stubdir ⟨dir⟩* (page 203) and *-dumpdir ⟨dir⟩* (page 203).

-osuf ⟨suffix⟩
The `-osuf` ⟨suffix⟩ will change the `.o` file suffix for object files to whatever you specify. We use this when compiling libraries, so that objects for the profiling versions of the libraries don't clobber the normal ones.

-dynosuf ⟨suffix⟩
When using `-dynamic-too`, option `-dynosuf` ⟨suffix⟩ is the counterpart of `-osuf`. It changes the `.dyn_o` file suffix for dynamic object files.

-hisuf ⟨suffix⟩
Similarly, the `-hisuf` ⟨suffix⟩ will change the `.hi` file suffix for non-system interface files

(see *Other options related to interface files* (page 205)).

The `-hisuf`/`-osuf` game is particularly useful if you want to compile a program both with and without profiling, in the same directory. You can say:

```
ghc ...
```

to get the ordinary version, and

```
ghc ... -osuf prof.o -hisuf prof.hi -prof -fprof-auto
```

to get the profiled version.

-dynhisuf ⟨suffix⟩
When using `-dynamic-too`, option `-dynhisuf` ⟨suffix⟩ is the counterpart of `-hisuf`. It changes the `.dyn_hi` file suffix for dynamic interface files.

-hiesuf ⟨suffix⟩
The `-hiesuf` ⟨suffix⟩ will change the `.hie` file suffix for extended interface files to whatever you specify.

-hcsuf ⟨suffix⟩
Finally, the option `-hcsuf` ⟨suffix⟩ will change the `.hc` file suffix for compiler-generated intermediate C files.

5.8.5 Keeping Intermediate Files

The following options are useful for keeping (or not keeping) certain intermediate files around, when normally GHC would throw these away after compilation:

-keep-hc-file
-keep-hc-files
Keep intermediate `.hc` files when doing `.hs`-to-`.o` compilations via *C* (page 237) (Note: `.hc` files are only generated by *unregisterised* (page 237) compilers).

-keep-hi-files
Keep intermediate `.hi` files. This is the default. You may use `-no-keep-hi-files` if you are not interested in the `.hi` files.

-keep-hscpp-file
-keep-hscpp-files
Keep the output of the `CPP` pre-processor phase as `.hscpp` files. A `.hscpp` file is only created, if a module gets compiled and uses the C pre-processor.

-keep-llvm-file
-keep-llvm-files

Implies *-fllvm* (page 243)

Keep intermediate `.ll` files when doing `.hs`-to-`.o` compilations via *LLVM* (page 236) (Note: `.ll` files aren't generated when using the native code generator, you may need to use *-fllvm* (page 243) to force them to be produced).

-keep-o-files
Keep intermediate `.o` files. This is the default. You may use `-no-keep-o-files` if you are not interested in the `.o` files.

-keep-s-file
-keep-s-files
Keep intermediate `.s` files.

-keep-tmp-files
: Instructs the GHC driver not to delete any of its temporary files, which it normally keeps in `/tmp` (or possibly elsewhere; see *Redirecting temporary files* (page 205)). Running GHC with `-v` will show you what temporary files were generated along the way.

5.8.6 Redirecting temporary files

-tmpdir ⟨dir⟩
: If you have trouble because of running out of space in `/tmp` (or wherever your installation thinks temporary files should go), you may use the *-tmpdir ⟨dir⟩* (page 205) option to specify an alternate directory. For example, `-tmpdir .` says to put temporary files in the current working directory.

  Alternatively, use your `TMPDIR` environment variable. Set it to the name of the directory where temporary files should be put. GCC and other programs will honour the `TMPDIR` variable as well.

5.8.7 Other options related to interface files

-ddump-hi
: Dumps the new interface to standard output.

-ddump-hi-diffs
: The compiler does not overwrite an existing `.hi` interface file if the new one is the same as the old one; this is friendly to **make**. When an interface does change, it is often enlightening to be informed. The *-ddump-hi-diffs* (page 205) option will make GHC report the differences between the old and new `.hi` files.

-ddump-minimal-imports
: Dump to the file `M.imports` (where ⟨M⟩ is the name of the module being compiled) a "minimal" set of import declarations. The directory where the `.imports` files are created can be controlled via the *-dumpdir ⟨dir⟩* (page 203) option.

  You can safely replace all the import declarations in `M.hs` with those found in its respective `.imports` file. Why would you want to do that? Because the "minimal" imports (a) import everything explicitly, by name, and (b) import nothing that is not required. It can be quite painful to maintain this property by hand, so this flag is intended to reduce the labour.

--show-iface ⟨file⟩
: where ⟨file⟩ is the name of an interface file, dumps the contents of that interface in a human-readable format. See *Modes of operation* (page 68).

5.8.8 Options related to extended interface files

GHC builds up a wealth of information about a Haskell source file as it compiles it. Extended interface files are a way of persisting some of this information to disk so that external tools, such as IDE's, can avoid parsing, typechecking, and renaming all over again. These files contain

- a simplified AST
  - nodes are annotated with source positions and types
  - identifiers are annotated with scope information

- the raw bytes of the initial Haskell source

The GHC API exposes functions for reading and writing these files.

-fwrite-ide-info
: Writes out extended interface files alongside regular interface files. Just like regular interface files, GHC has a recompilation check to detect out of date or missing extended interface files.

-fvalidate-ide-info
: Runs a series of sanity checks and lints on the extended interface files that are being written out. These include testing things properties such as variables not occurring outside of their expected scopes.

The format in which GHC currently stores its typechecked AST, makes it costly to collect the types for some expressions nodes. For the sake of performance, GHC currently chooses to skip over these, so not all expression nodes should be expected to have type information on them. See #16233 for more.

5.8.9 The recompilation checker

-fforce-recomp
: Turn off recompilation checking (which is on by default). Recompilation checking normally stops compilation early, leaving an existing `.o` file in place, if it can be determined that the module does not need to be recompiled.

-fignore-optim-changes

-fignore-hpc-changes

In the olden days, GHC compared the newly-generated `.hi` file with the previous version; if they were identical, it left the old one alone and didn't change its modification date. In consequence, importers of a module with an unchanged output `.hi` file were not recompiled.

This doesn't work any more. Suppose module `C` imports module `B`, and `B` imports module `A`. So changes to module `A` might require module `C` to be recompiled, and hence when `A.hi` changes we should check whether `C` should be recompiled. However, the dependencies of `C` will only list `B.hi`, not `A.hi`, and some changes to `A` (changing the definition of a function that appears in an inlining of a function exported by `B`, say) may conceivably not change `B.hi` one jot. So now…

GHC calculates a fingerprint (in fact an MD5 hash) of each interface file, and of each declaration within the interface file. It also keeps in every interface file a list of the fingerprints of everything it used when it last compiled the file. If the MD5 hash of the source file stored in the `.hi` file hasn't changed, the `.o` file's modification date is greater than or equal to that of the `.hi` file, and the recompilation checking is on, GHC will be clever. It compares the fingerprints on the things it needs this time with the fingerprints on the things it needed last time (gleaned from the interface file of the module being compiled); if they are all the same it stops compiling early in the process saying "Compilation IS NOT required". What a beautiful sight!

You can read about how all this works in the GHC commentary.

5.8.9.1 Recompilation for Template Haskell and Plugins

Recompilation checking gets a bit more complicated when using Template Haskell or plugins. Both these features execute code at compile time and so if any of the executed code changes then it's necessary to recompile the module. Consider the top-level splice:

```
main = $(foo bar [| () |])
```

When the module is compiled `foo bar [| () |]` will be evaluated and the resulting code placed into the program. The dependencies of the expression are calculated and stored during module compilation. When the interface file is written, additional dependencies are created on the object file dependencies of the expression. For instance, if `foo` is from module `A` and `bar` is from module `B`, the module will now depend on `A.o` and `B.o`, if either of these change then the module will be recompiled.

5.8.10 How to compile mutually recursive modules

GHC supports the compilation of mutually recursive modules. This section explains how.

Every cycle in the module import graph must be broken by a `hs-boot` file. Suppose that modules `A.hs` and `B.hs` are Haskell source files, thus:

```
module A where
    import B( TB(..) )

    newtype TA = MkTA Int

    f :: TB -> TA
    f (MkTB x) = MkTA x

module B where
    import {-# SOURCE #-} A( TA(..) )

    data TB = MkTB !Int

    g :: TA -> TB
    g (MkTA x) = MkTB x
```

Here `A` imports `B`, but `B` imports `A` with a `{-# SOURCE #-}` pragma, which breaks the circular dependency. Every loop in the module import graph must be broken by a `{-# SOURCE #-}` import; or, equivalently, the module import graph must be acyclic if `{-# SOURCE #-}` imports are ignored.

For every module `A.hs` that is `{-# SOURCE #-}`-imported in this way there must exist a source file `A.hs-boot`. This file contains an abbreviated version of `A.hs`, thus:

```
module A where
    newtype TA = MkTA Int
```

To compile these three files, issue the following commands:

```
ghc -c A.hs-boot    -- Produces A.hi-boot, A.o-boot
ghc -c B.hs         -- Consumes A.hi-boot, produces B.hi, B.o
ghc -c A.hs         -- Consumes B.hi, produces A.hi, A.o
ghc -o foo A.o B.o  -- Linking the program
```

There are several points to note here:

- The file `A.hs-boot` is a programmer-written source file. It must live in the same directory as its parent source file `A.hs`. Currently, if you use a literate source file `A.lhs` you must also use a literate boot file, `A.lhs-boot`; and vice versa.
- A `hs-boot` file is compiled by GHC, just like a `hs` file:

  ```
  ghc -c A.hs-boot
  ```

  When a hs-boot file `A.hs-boot` is compiled, it is checked for scope and type errors. When its parent module `A.hs` is compiled, the two are compared, and an error is reported if the two are inconsistent.
- Just as compiling `A.hs` produces an interface file `A.hi`, and an object file `A.o`, so compiling `A.hs-boot` produces an interface file `A.hi-boot`, and a pseudo-object file `A.o-boot`:
  - The pseudo-object file `A.o-boot` is empty (don't link it!), but it is very useful when using a Makefile, to record when the `A.hi-boot` was last brought up to date (see *Using make* (page 215)).
  - The `hi-boot` generated by compiling a `hs-boot` file is in the same machine-generated binary format as any other GHC-generated interface file (e.g. `B.hi`). You can display its contents with `ghc --show-iface`. If you specify a directory for interface files, the `-hidir` flag, then that affects `hi-boot` files too.
- If hs-boot files are considered distinct from their parent source files, and if a `{-# SOURCE #-}` import is considered to refer to the hs-boot file, then the module import graph must have no cycles. The command `ghc -M` will report an error if a cycle is found.
- A module `M` that is `{-# SOURCE #-}`-imported in a program will usually also be ordinarily imported elsewhere. If not, `ghc --make` automatically adds `M` to the set of modules it tries to compile and link, to ensure that `M`'s implementation is included in the final program.

A hs-boot file need only contain the bare minimum of information needed to get the bootstrapping process started. For example, it doesn't need to contain declarations for *everything* that module `A` exports, only the things required by the module(s) that import `A` recursively.

A hs-boot file is written in a subset of Haskell:

- The module header (including the export list), and import statements, are exactly as in Haskell, and so are the scoping rules. Hence, to mention a non-Prelude type or class, you must import it.
- There must be no value declarations, but there can be type signatures for values. For example:

  ```
  double :: Int -> Int
  ```

- Fixity declarations are exactly as in Haskell.
- Vanilla type synonym declarations are exactly as in Haskell.
- Open type and data family declarations are exactly as in Haskell.
- A closed type family may optionally omit its equations, as in the following example:

  ```
  type family ClosedFam a where ..
  ```

  The `..` is meant literally – you should write two dots in your file. Note that the `where` clause is still necessary to distinguish closed families from open ones. If you give any equations of a closed family, you must give all of them, in the same order as they appear in the accompanying Haskell file.

- A data type declaration can either be given in full, exactly as in Haskell, or it can be given abstractly, by omitting the '=' sign and everything that follows. For example:

  ```
  data T a b
  ```

  In a *source* program this would declare TA to have no constructors (a GHC extension: see *Data types with no constructors* (page 321)), but in an hi-boot file it means "I don't know or care what the constructors are". This is the most common form of data type declaration, because it's easy to get right. You *can* also write out the constructors but, if you do so, you must write it out precisely as in its real definition.

  If you do not write out the constructors, you may need to give a kind annotation (*Explicitly-kinded quantification* (page 511)), to tell GHC the kind of the type variable, if it is not "*". (In source files, this is worked out from the way the type variable is used in the constructors.) For example:

  ```
  data R (x :: * -> *) y
  ```

  You cannot use `deriving` on a data type declaration; write an `instance` declaration instead.

- Class declarations can either be given in full, exactly as in Haskell, or they can be given abstractly by omitting everything other than the instance head: no superclasses, no class methods, no associated types. However, if the class has any ::extension::*FunctionalDependencies*, those given in the hs-boot file must be the same.

  If the class declaration is given in full, the entire class declaration must be identical, up to a renaming of the type variables bound by the class head. This means:

  - The class head must be the same.
  - The class context must be the same, up to simplification of constraints.
  - If there are any ::extension::*FunctionalDependencies*, these must be the same.
  - The order, names, and types of the class methods must be the same.
  - The arity and kinds of any associated types must be the same.
  - Default methods as well as default signatures (see ::extension::*DefaultSignatures*) must be provided for the same methods, and must be the same.
  - Default declarations for associated types must be provided for the same types, and must be the same.

  To declare a class with no methods in an hs-boot file, it must have a superclass. If the class has no superclass constraints, add an empty one, e.g.

  ```
  class () => C a
  ```

  This is a full class declaration, not an abstract declaration in which the methods were omitted.

- You can include instance declarations just as in Haskell; but omit the "where" part.
- The default role for abstract datatype parameters is now representational. (An abstract datatype is one with no constructors listed.) To get another role, use a role annotation. (See *Roles* (page 417).)

5.8.11 Module signatures

GHC 8.2 supports module signatures (`hsig` files), which allow you to write a signature in place of a module implementation, deferring the choice of implementation until a later point in time. This feature is not intended to be used without Cabal; this manual entry will focus on the syntax and semantics of signatures.

To start with an example, suppose you had a module `A` which made use of some string operations. Using normal module imports, you would only be able to pick a particular implementation of strings:

```
module Str where
    type Str = String

    empty :: Str
    empty = ""

    toString :: Str -> String
    toString s = s

module A where
    import Str
    z = toString empty
```

By replacing `Str.hs` with a signature `Str.hsig`, `A` (and any other modules in this package) are now parametrized by a string implementation:

```
signature Str where
    data Str
    empty :: Str
    toString :: Str -> String
```

We can typecheck `A` against this signature, or we can instantiate `Str` with a module that provides the following declarations. Refer to Cabal's documentation for a more in-depth discussion on how to instantiate signatures.

Module signatures actually consist of two closely related features:

- The ability to define an `hsig` file, containing type definitions and type signature for values which can be used by modules that import the signature, and must be provided by the eventual implementing module, and
- The ability to *inherit* required signatures from packages we depend upon, combining the signatures into a single merged signature which reflects the requirements of any locally defined signature, as well as the requirements of our dependencies.

A signature file is denoted by an `hsig` file; every required signature must have an `hsig` file (even if it is an empty one), including required signatures inherited from dependencies. Signatures can be imported using an ordinary `import Sig` declaration.

`hsig` files are written in a variant of Haskell similar to `hs-boot` files, but with some slight changes:

- The header of a signature is `signature A where ...` (instead of the usual `module A where ...`).
- Import statements and scoping rules are exactly as in Haskell. To mention a non-Prelude type or class, you must import it.

- Unlike regular modules, the defined entities of a signature include not only those written in the local `hsig` file, but also those from inherited signatures (as inferred from the *-package-id (unit-id)* (page 221) flags). These entities are not considered in scope when typechecking the local `hsig` file, but are available for import by any module or signature which imports the signature. The one exception to this rule is the export list, described below.

  If a declaration occurs in multiple inherited signatures, they will be *merged* together. For values, we require that the types from both signatures match exactly; however, other declarations may merge in more interesting ways. The merging operation in these cases has the effect of textually replacing all occurrences of the old name with a reference to the new, merged declaration. For example, if we have the following two signatures:

  ```
  signature A where
      data T
      f :: T -> T

  signature A where
      data T = MkT
      g :: T
  ```

  the resulting merged signature would be:

  ```
  signature A where
      data T = MkT
      f :: T -> T
      g :: T
  ```

- If no export list is provided for a signature, the exports of a signature are all of its defined entities merged with the exports of all inherited signatures.

  If you want to reexport an entity from a signature, you must also include a `module SigName` export, so that all of the entities defined in the signature are exported. For example, the following module exports both `f` and `Int` from `Prelude`:

  ```
  signature A(module A, Int) where
      import Prelude (Int)
      f :: Int
  ```

  Reexports merge with local declarations; thus, the signature above would successfully merge with:

  ```
  signature A where
      data Int
  ```

  The only permissible implementation of such a signature is a module which reexports precisely the same entity:

  ```
  module A (f, Int) where
      import Prelude (Int)
      f = 2 :: Int
  ```

  Conversely, any entity requested by a signature can be provided by a reexport from the implementing module. This is different from `hs-boot` files, which require every entity to be defined locally in the implementing module.

- GHC has experimental support for *signature thinning*, which is used when a signature has an explicit export list without a module export of the signature itself. In this case, the export list applies to the final export list *after* merging, in particular, you may refer to entities which are not declared in the body of the local `hsig` file.

The semantics in this case is that the set of required entities is defined exclusively by its exports; if an entity is not mentioned in the export list, it is not required. The motivation behind this feature is to allow a library author to provide an omnibus signature containing the type of every function someone might want to use, while a client thins down the exports to the ones they actually require. For example, supposing that you have inherited a signature for strings, you might write a local signature of this form, listing only the entities that you need:

```
signature Str (Str, empty, append, concat) where
    -- empty
```

A few caveats apply here. First, it is illegal to export an entity which refers to a locally defined type which itself is not exported (GHC will report an error in this case). Second, signatures which come from dependencies which expose modules cannot be thinned in this way (after all, the dependency itself may need the entity); these requirements are unconditionally exported. Finally, any module reexports must refer to modules imported by the local signature (even if an inherited signature exported the module).

We may change the syntax and semantics of this feature in the future.

- The declarations and types from signatures of dependencies that will be merged in are not in scope when type checking an `hsig` file. To refer to any such type, you must declare it yourself:

```
-- OK, assuming we inherited an A that defines T
signature A (T) where
    -- empty

-- Not OK
signature A (T, f) where
    f :: T -> T

-- OK
signature A (T, f) where
    data T
    f :: T -> T
```

- There must be no value declarations, but there can be type signatures for values. For example, we might define the signature:

```
signature A where
    double :: Int -> Int
```

A module implementing `A` would have to export the function `double` with a type definitionally equal to the signature. Note that this means you can't implement `double` using a polymorphic function `double :: Num a => a -> a`.

Note that signature matching does check if *fixity* matches, so be sure specify fixity of ordinary identifiers if you intend to use them with backticks.

- Fixity, type synonym, open type/data family declarations are permitted as in normal Haskell.
- Closed type family declarations are permitted as in normal Haskell. They can also be given abstractly, as in the following example:

```
type family ClosedFam a where ..
```

The `..` is meant literally – you should write two dots in your file. The `where` clause distinguishes closed families from open ones.

- A data type declaration can either be given in full, exactly as in Haskell, or it can be given abstractly, by omitting the '=' sign and everything that follows. For example:

```
signature A where
    data T a b
```

Abstract data types can be implemented not only with data declarations, but also newtypes and type synonyms (with the restriction that a type synonym must be fully eta-reduced, e.g., `type T = ...` to be accepted.) For example, the following are all valid implementations of the T above:

```
-- Algebraic data type
data T a b = MkT a b

-- Newtype
newtype T a b = MkT (a, b)

-- Type synonym
data T2 a b = MkT2 a a b b
type T = T2
```

Data type declarations merge only with other data type declarations which match exactly, except abstract data, which can merge with `data`, `newtype` or `type` declarations. Merges with type synonyms are especially useful: suppose you are using a package of strings which has left the type of characters in the string unspecified:

```
signature Str where
    data Str
    data Elem
    head :: Str -> Elem
```

If you locally define a signature which specifies `type Elem = Char`, you can now use `head` from the inherited signature as if it returned a `Char`.

If you do not write out the constructors, you may need to give a kind to tell GHC what the kinds of the type variables are, if they are not the default *. Unlike regular data type declarations, the return kind of an abstract data declaration can be anything (in which case it probably will be implemented using a type synonym.) This can be used to allow compile-time representation polymorphism (as opposed to *run-time representation polymorphism* (page **??**)), as in this example:

```
signature Number where
    import GHC.Types
    data Rep :: RuntimeRep
    data Number :: TYPE Rep
    plus :: Number -> Number -> Number
```

Roles of type parameters are subject to the subtyping relation `phantom < representational < nominal`: for example, an abstract type with a nominal type parameter can be implemented using a concrete type with a representational type parameter. Merging respects this subtyping relation (e.g., `nominal` merged with `representational` is `representational`.) Roles in signatures default to `nominal`, which gives maximum flexibility on the implementor's side. You should only need to give an explicit role annotation if a client of the signature would like to coerce the abstract type in a type parameter (in which case you should specify `representational` explicitly.) Unlike regular data types, we do *not* assume that abstract data types are representationally injective: if we have `Coercible (T a) (T b)`, and `T` has role `nominal`, this does not imply that `a ~ b`.

- A class declarations can either be abstract or concrete. An abstract class is one with no superclasses or class methods:

```
signature A where
    class Key k
```

It can be implemented in any way, with any set of superclasses and methods; however, modules depending on an abstract class are not permitted to define instances (as of GHC 8.2, this restriction is not checked, see #13086.) These declarations can be implemented by type synonyms of kind `Constraint`; this can be useful if you want to parametrize over a constraint in functions. For example, with the `ConstraintKinds` extension, this type synonym is a valid implementation of the signature above:

```
module A where
    type Key = Eq
```

A concrete class specifies its superclasses, methods, default method signatures (but not their implementations) and a `MINIMAL` pragma. Unlike regular Haskell classes, you don't have to explicitly declare a default for a method to make it optional vis-a-vis the `MINIMAL` pragma.

When merging class declarations, we require that the superclasses and methods match exactly; however, `MINIMAL` pragmas are logically ORed together, and a method with a default signature will merge successfully against one that does not.

- You can include instance declarations as in Haskell; just omit the "where" part. An instance declaration need not be implemented directly; if an instance can be derived based on instances in the environment, it is considered implemented. For example, the following signature:

```
signature A where
    data Str
    instance Eq Str
```

is considered implemented by the following module, since there are instances of `Eq` for `[]` and `Char` which can be combined to form an instance `Eq [Char]`:

```
module A where
    type Str = [Char]
```

Unlike other declarations, for which only the entities declared in a signature file are brought into scope, instances from the implementation are always brought into scope, even if they were not declared in the signature file. This means that a module may typecheck against a signature, but not against a matching implementation. You can avoid situations like this by never defining orphan instances inside a package that has signatures.

Instance declarations are only merged if their heads are exactly the same, so it is possible to get into a situation where GHC thinks that instances in a signature are overlapping, even if they are implemented in a non-overlapping way. If this is giving you problems give us a shout.

- Any orphan instances which are brought into scope by an import from a signature are unconditionally considered in scope, even if the eventual implementing module doesn't actually import the same orphans.

Known limitations:

- Pattern synonyms are not supported.
- Algebraic data types specified in a signature cannot be implemented using pattern synonyms. See #12717

5.8.12 Using make

It is reasonably straightforward to set up a `Makefile` to use with GHC, assuming you name your source files the same as your modules. Thus:

```
HC      = ghc
HC_OPTS = -cpp $(EXTRA_HC_OPTS)

SRCS = Main.lhs Foo.lhs Bar.lhs
OBJS = Main.o   Foo.o   Bar.o

.SUFFIXES : .o .hs .hi .lhs .hc .s

cool_pgm : $(OBJS)
        rm -f $@
        $(HC) -o $@ $(HC_OPTS) $(OBJS)

# Standard suffix rules
.o.hi:
        @:

.lhs.o:
        $(HC) -c $< $(HC_OPTS)

.hs.o:
        $(HC) -c $< $(HC_OPTS)

.o-boot.hi-boot:
        @:

.lhs-boot.o-boot:
        $(HC) -c $< $(HC_OPTS)

.hs-boot.o-boot:
        $(HC) -c $< $(HC_OPTS)

# Inter-module dependencies
Foo.o Foo.hc Foo.s    : Baz.hi          # Foo imports Baz
Main.o Main.hc Main.s : Foo.hi Baz.hi   # Main imports Foo and Baz
```

Note: Sophisticated **make** variants may achieve some of the above more elegantly. Notably, **gmake**'s pattern rules let you write the more comprehensible:

```
%.o : %.lhs
        $(HC) -c $< $(HC_OPTS)
```

What we've shown should work with any make.

Note the cheesy .o.hi rule: It records the dependency of the interface (.hi) file on the source. The rule says a .hi file can be made from a .o file by doing…nothing. Which is true.

Note that the suffix rules are all repeated twice, once for normal Haskell source files, and once for hs-boot files (see *How to compile mutually recursive modules* (page 207)).

Note also the inter-module dependencies at the end of the Makefile, which take the form

```
Foo.o Foo.hc Foo.s    : Baz.hi          # Foo imports Baz
```

They tell make that if any of Foo.o, Foo.hc or Foo.s have an earlier modification date than Baz.hi, then the out-of-date file must be brought up to date. To bring it up to date, make looks for a rule to do so; one of the preceding suffix rules does the job nicely. These dependencies can be generated automatically by ghc; see *Dependency generation* (page 216)

5.8.13 Dependency generation

Putting inter-dependencies of the form Foo.o : Bar.hi into your Makefile by hand is rather error-prone. Don't worry, GHC has support for automatically generating the required dependencies. Add the following to your Makefile:

```
depend :
        ghc -M $(HC_OPTS) $(SRCS)
```

Now, before you start compiling, and any time you change the imports in your program, do make depend before you do make cool_pgm. The command ghc -M will append the needed dependencies to your Makefile.

In general, ghc -M Foo does the following. For each module M in the set Foo plus all its imports (transitively), it adds to the Makefile:

- A line recording the dependence of the object file on the source file.

  ```
  M.o : M.hs
  ```

  (or M.lhs if that is the filename you used).

- For each import declaration import X in M, a line recording the dependence of M on X:

  ```
  M.o : X.hi
  ```

- For each import declaration import {-# SOURCE #-} X in M, a line recording the dependence of M on X:

  ```
  M.o : X.hi-boot
  ```

  (See *How to compile mutually recursive modules* (page 207) for details of hi-boot style interface files.)

If M imports multiple modules, then there will be multiple lines with M.o as the target.

There is no need to list all of the source files as arguments to the ghc -M command; ghc traces the dependencies, just like ghc --make (a new feature in GHC 6.4).

Note that `ghc -M` needs to find a *source file* for each module in the dependency graph, so that it can parse the import declarations and follow dependencies. Any pre-compiled modules without source files must therefore belong to a package[1].

By default, `ghc -M` generates all the dependencies, and then concatenates them onto the end of `makefile` (or `Makefile` if `makefile` doesn't exist) bracketed by the lines "`# DO NOT DELETE: Beginning of Haskell dependencies`" and "`# DO NOT DELETE: End of Haskell dependencies`". If these lines already exist in the `makefile`, then the old dependencies are deleted first.

Don't forget to use the same `-package` options on the `ghc -M` command line as you would when compiling; this enables the dependency generator to locate any imported modules that come from packages. The package modules won't be included in the dependencies generated, though (but see the `-include-pkg-deps` option below).

The dependency generation phase of GHC can take some additional options, which you may find useful. The options which affect dependency generation are:

`-ddump-mod-cycles`
: Display a list of the cycles in the module graph. This is useful when trying to eliminate such cycles.

`-v2`
: Print a full list of the module dependencies to stdout. (This is the standard verbosity flag, so the list will also be displayed with `-v3` and `-v4`; see *Verbosity options* (page 76).)

`-dep-makefile` ⟨file⟩
: Use ⟨file⟩ as the makefile, rather than `makefile` or `Makefile`. If ⟨file⟩ doesn't exist, `mkdependHS` creates it. We often use `-dep-makefile .depend` to put the dependencies in `.depend` and then `include` the file `.depend` into `Makefile`.

`-dep-suffix` ⟨suffix⟩
: Make dependencies that declare that files with suffix `.⟨suf⟩⟨osuf⟩` depend on interface files with suffix `.⟨suf⟩hi`, or (for `{-# SOURCE #-}` imports) on `.hi-boot`. Multiple `-dep-suffix` flags are permitted. For example, `-dep-suffix a_ -dep-suffix b_` will make dependencies for `.hs` on `.hi`, `.a_hs` on `.a_hi`, and `.b_hs` on `.b_hi`. If you do not use this flag then the empty suffix is used.

`-exclude-module`=⟨file⟩
: Regard `⟨file⟩` as "stable"; i.e., exclude it from having dependencies on it.

`-include-pkg-deps`
: Regard modules imported from packages as unstable, i.e., generate dependencies on any imported package modules (including `Prelude`, and all other standard Haskell libraries). Dependencies are not traced recursively into packages; dependencies are only generated for home-package modules on external-package modules directly imported by the home package module. This option is normally only used by the various system libraries.

`-include-cpp-deps`
: Output preprocessor dependencies. This only has an effect when the CPP language extension is enabled. These dependencies are files included with the `#include` preprocessor directive (as well as transitive includes) and implicitly included files such as standard c preprocessor headers and a GHC version header. One exception to this is that GHC generates a temporary header file (during compilation) containing package version macros. As this is only a temporary file that GHC will always generate, it is not output as a dependency.

[1] This is a change in behaviour relative to 6.2 and earlier.

5.8.14 Orphan modules and instance declarations

Haskell specifies that when compiling module `M`, any instance declaration in any module "below" `M` is visible. (Module `A` is "below" `M` if `A` is imported directly by `M`, or if `A` is below a module that `M` imports directly.) In principle, GHC must therefore read the interface files of every module below `M`, just in case they contain an instance declaration that matters to `M`. This would be a disaster in practice, so GHC tries to be clever.

In particular, if an instance declaration is in the same module as the definition of any type or class mentioned in the *head* of the instance declaration (the part after the "=>"; see *Instance termination rules* (page 483)), then GHC has to visit that interface file anyway. Example:

```
module A where
  instance C a => D (T a) where ...
  data T a = ...
```

The instance declaration is only relevant if the type `T` is in use, and if so, GHC will have visited `A`'s interface file to find `T`'s definition.

The only problem comes when a module contains an instance declaration and GHC has no other reason for visiting the module. Example:

```
module Orphan where
  instance C a => D (T a) where ...
  class C a where ...
```

Here, neither `D` nor `T` is declared in module `Orphan`. We call such modules "orphan modules". GHC identifies orphan modules, and visits the interface file of every orphan module below the module being compiled. This is usually wasted work, but there is no avoiding it. You should therefore do your best to have as few orphan modules as possible.

Functional dependencies complicate matters. Suppose we have:

```
module B where
  instance E T Int where ...
  data T = ...
```

Is this an orphan module? Apparently not, because `T` is declared in the same module. But suppose class `E` had a functional dependency:

```
module Lib where
  class E x y | y -> x where ...
```

Then in some importing module `M`, the constraint `(E a Int)` should be "improved" by setting `a = T`, *even though there is no explicit mention* of `T` in `M`.

These considerations lead to the following definition of an orphan module:

- An *orphan module* orphan module contains at least one *orphan instance* or at least one *orphan rule*.
- An instance declaration in a module `M` is an *orphan instance* if
  - The class of the instance declaration is not declared in `M`, and
  - *Either* the class has no functional dependencies, and none of the type constructors in the instance head is declared in `M`; *or* there is a functional dependency for which none of the type constructors mentioned in the *non-determined* part of the instance head is defined in `M`.

Only the instance head counts. In the example above, it is not good enough for C's declaration to be in module A; it must be the declaration of D or T.

- A rewrite rule in a module M is an *orphan rule* orphan rule if none of the variables, type constructors, or classes that are free in the left hand side of the rule are declared in M.

If you use the flag *-Worphans* (page 99), GHC will warn you if you are creating an orphan module. Like any warning, you can switch the warning off with *-Wno-orphans* (page 99), and *-Werror* (page 87) will make the compilation fail if the warning is issued.

You can identify an orphan module by looking in its interface file, M.hi, using the *--show-iface ⟨file⟩* (page 69) *mode* (page 68). If there is a [orphan module] on the first line, GHC considers it an orphan module.

5.9 Packages

A package is a library of Haskell modules known to the compiler. GHC comes with several packages: see the accompanying library documentation. More packages to install can be obtained from HackageDB.

Using a package couldn't be simpler: if you're using --make or GHCi, then most of the installed packages will be automatically available to your program without any further options. The exceptions to this rule are covered below in *Using Packages* (page 219).

Building your own packages is also quite straightforward: we provide the Cabal infrastructure which automates the process of configuring, building, installing and distributing a package. All you need to do is write a simple configuration file, put a few files in the right places, and you have a package. See the Cabal documentation for details, and also the Cabal libraries (Distribution.Simple, for example).

5.9.1 Using Packages

GHC only knows about packages that are *installed*. Installed packages live in package databases. For details on package databases and how to control which package databases or specific set of packages are visible to GHC, see *Package Databases* (page 224).

To see which packages are currently available, use the ghc-pkg list command:

```
$ ghc-pkg list
/usr/lib/ghc-6.12.1/package.conf.d:
    Cabal-1.7.4
    array-0.2.0.1
    base-3.0.3.0
    base-4.2.0.0
    bin-package-db-0.0.0.0
    binary-0.5.0.1
    bytestring-0.9.1.4
    containers-0.2.0.1
    directory-1.0.0.2
    (dph-base-0.4.0)
    (dph-par-0.4.0)
    (dph-prim-interface-0.4.0)
    (dph-prim-par-0.4.0)
    (dph-prim-seq-0.4.0)
    (dph-seq-0.4.0)
```

(continues on next page)

(continued from previous page)

```
    extensible-exceptions-0.1.1.0
    ffi-1.0
    filepath-1.1.0.1
    (ghc-6.12.1)
    ghc-prim-0.1.0.0
    haskeline-0.6.2
    haskell98-1.0.1.0
    hpc-0.5.0.2
    integer-gmp-0.1.0.0
    mtl-1.1.0.2
    old-locale-1.0.0.1
    old-time-1.0.0.1
    pretty-1.0.1.0
    process-1.0.1.1
    random-1.0.0.1
    rts-1.0.1
    syb-0.1.0.0
    template-haskell-2.4.0.0
    terminfo-0.3.1
    time-1.1.4
    unix-2.3.1.0
    utf8-string-0.3.4
```

An installed package is either *exposed* or *hidden* by default. Packages hidden by default are listed in parentheses (e.g. `(lang-1.0)`), or possibly in blue if your terminal supports colour, in the output of `ghc-pkg list`. Command-line flags, described below, allow you to expose a hidden package or hide an exposed one. Only modules from exposed packages may be imported by your Haskell code; if you try to import a module from a hidden package, GHC will emit an error message. It should be noted that a hidden package might still get linked with your program as a dependency of an exposed package, it is only restricted from direct imports.

If there are multiple exposed versions of a package, GHC will prefer the latest one. Additionally, some packages may be broken: that is, they are missing from the package database, or one of their dependencies are broken; in this case; these packages are excluded from the default set of packages.

Note: If you're using Cabal, then the exposed or hidden status of a package is irrelevant: the available packages are instead determined by the dependencies listed in your `.cabal` specification. The exposed/hidden status of packages is only relevant when using `ghc` or `ghci` directly.

Similar to a package's hidden status is a package's trusted status. A package can be either trusted or not trusted (distrusted). By default packages are distrusted. This property of a package only plays a role when compiling code using GHC's Safe Haskell feature (see *Safe Haskell* (page 577)) with the `-fpackage-trust` flag enabled.

To see which modules are provided by a package use the `ghc-pkg` command (see *Package management (the ghc-pkg command)* (page 228)):

```
$ ghc-pkg field network exposed-modules
exposed-modules: Network.BSD,
                 Network.CGI,
                 Network.Socket,
```

(continues on next page)

(continued from previous page)

```
                Network.URI,
                Network
```

The GHC command line options that control packages are:

-package ⟨pkg⟩
This option causes the installed package ⟨pkg⟩ to be exposed. The package ⟨pkg⟩ can be specified in full with its version number (e.g. `network-1.0`) or the version number can be omitted in which case GHC will automatically expose the latest non-broken version from the installed versions of the package.

By default (when *-hide-all-packages* (page 221) is not specified), GHC exposes only one version of a package, all other versions become hidden. If *-package* option is specified multiple times for the same package the last one overrides the previous ones. On the other hand, if *-hide-all-packages* (page 221) is used, GHC allows you to expose multiple versions of a package by using the *-package* option multiple times with different versions of the same package.

-package supports thinning and renaming described in *Thinning and renaming modules* (page 223).

The `-package ⟨pkg⟩` option also causes package ⟨pkg⟩ to be linked into the resulting executable or shared object. Whether a packages' library is linked statically or dynamically is controlled by the flag pair *-static* (page 246)/ *-dynamic* (page 246).

In *--make* (page 68) mode and *--interactive* (page 68) mode (see *Modes of operation* (page 68)), the compiler normally determines which packages are required by the current Haskell modules, and links only those. In batch mode however, the dependency information isn't available, and explicit `-package` options must be given when linking. The one other time you might need to use `-package` to force linking a package is when the package does not contain any Haskell modules (it might contain a C library only, for example). In that case, GHC will never discover a dependency on it, so it has to be mentioned explicitly.

For example, to link a program consisting of objects `Foo.o` and `Main.o`, where we made use of the `network` package, we need to give GHC the `-package` flag thus:

```
$ ghc -o myprog Foo.o Main.o -package network
```

The same flag is necessary even if we compiled the modules from source, because GHC still reckons it's in batch mode:

```
$ ghc -o myprog Foo.hs Main.hs -package network
```

-package-id ⟨unit-id⟩
Exposes a package like *-package ⟨pkg⟩* (page 221), but the package is named by its unit ID (i.e. the value of `id` in its entry in the installed package database, also previously known as an installed package ID) rather than by name. This is a more robust way to name packages, and can be used to select packages that would otherwise be shadowed. Cabal passes `-package-id` flags to GHC. `-package-id` supports thinning and renaming described in *Thinning and renaming modules* (page 223).

-hide-all-packages
Ignore the exposed flag on installed packages, and hide them all by default. If you use this flag, then any packages you require (including `base`) need to be explicitly exposed using *-package ⟨pkg⟩* (page 221) options.

This is a good way to insulate your program from differences in the globally exposed packages, and being explicit about package dependencies is a Good Thing. Cabal always passes the `-hide-all-packages` flag to GHC, for exactly this reason.

`-hide-package` `⟨pkg⟩`
This option does the opposite of *`-package ⟨pkg⟩`* (page 221): it causes the specified package to be hidden, which means that none of its modules will be available for import by Haskell `import` directives.

Note that the package might still end up being linked into the final program, if it is a dependency (direct or indirect) of another exposed package.

`-ignore-package` `⟨pkg⟩`
Causes the compiler to behave as if package ⟨pkg⟩, and any packages that depend on ⟨pkg⟩, are not installed at all.

Saying `-ignore-package ⟨pkg⟩` is the same as giving *`-hide-package ⟨pkg⟩`* (page 222) flags for ⟨pkg⟩ and all the packages that depend on ⟨pkg⟩. Sometimes we don't know ahead of time which packages will be installed that depend on ⟨pkg⟩, which is when the *`-ignore-package ⟨pkg⟩`* (page 222) flag can be useful.

`-no-auto-link-packages`
By default, GHC will automatically link in the `base` and `rts` packages. This flag disables that behaviour.

`-this-unit-id` `⟨unit-id⟩`
Tells GHC that the module being compiled forms part of unit ID ⟨unit-id⟩; internally, these keys are used to determine type equality and linker symbols. As of GHC 8.0, unit IDs must consist solely of alphanumeric characters, dashes, underscores and periods. GHC reserves the right to interpret other characters in a special way in later releases.

`-trust` `⟨pkg⟩`
This option causes the install package ⟨pkg⟩ to be both exposed and trusted by GHC. This command functions in a very similar way to the *`-package ⟨pkg⟩`* (page 221) command but in addition sets the selected packages to be trusted by GHC, regardless of the contents of the package database. (see *Safe Haskell* (page 577)).

`-distrust` `⟨pkg⟩`
This option causes the install package ⟨pkg⟩ to be both exposed and distrusted by GHC. This command functions in a very similar way to the *`-package ⟨pkg⟩`* (page 221) command but in addition sets the selected packages to be distrusted by GHC, regardless of the contents of the package database. (see *Safe Haskell* (page 577)).

`-distrust-all-packages`
Ignore the trusted flag on installed packages, and distrust them by default. If you use this flag and Safe Haskell then any packages you require to be trusted (including `base`) need to be explicitly trusted using *`-trust ⟨pkg⟩`* (page 585) options. This option does not change the exposed/hidden status of a package, so it isn't equivalent to applying *`-distrust ⟨pkg⟩`* (page 585) to all packages on the system. (see *Safe Haskell* (page 577)).

5.9.2 The `main` package

Every complete Haskell program must define `main` in module `Main` in package `main`. Omitting the *-this-unit-id ⟨unit-id⟩* (page 222) flag compiles code for package `main`. Failure to do so leads to a somewhat obscure link-time error of the form:

```
/usr/bin/ld: Undefined symbols:
_ZCMain_main_closure
```

5.9.3 Consequences of packages for the Haskell language

It is possible that by using packages you might end up with a program that contains two modules with the same name: perhaps you used a package `P` that has a *hidden* module `M`, and there is also a module `M` in your program. Or perhaps the dependencies of packages that you used contain some overlapping modules. Perhaps the program even contains multiple versions of a certain package, due to dependencies from other packages.

None of these scenarios gives rise to an error on its own[1], but they may have some interesting consequences. For instance, if you have a type `M.T` from version 1 of package `P`, then this is *not* the same as the type `M.T` from version 2 of package `P`, and GHC will report an error if you try to use one where the other is expected.

Formally speaking, in Haskell 98, an entity (function, type or class) in a program is uniquely identified by the pair of the module name in which it is defined and its name. In GHC, an entity is uniquely defined by a triple: package, module, and name.

5.9.4 Thinning and renaming modules

When incorporating packages from multiple sources, you may end up in a situation where multiple packages publish modules with the same name. Previously, the only way to distinguish between these modules was to use *Package-qualified imports* (page 319). However, since GHC 7.10, the *-package ⟨pkg⟩* (page 221) flags (and their variants) have been extended to allow a user to explicitly control what modules a package brings into scope, by analogy to the import lists that users can attach to module imports.

The basic syntax is that instead of specifying a package name P to the package flag `-package`, instead we specify both a package name and a parenthesized, comma-separated list of module names to import. For example, `-package "base (Data.List, Data.Bool)"` makes only `Data.List` and `Data.Bool` visible from package `base`. We also support renaming of modules, in case you need to refer to both modules simultaneously; this is supporting by writing `OldModName as NewModName`, e.g. `-package "base (Data.Bool as Bool)`. You can also write `-package "base with (Data.Bool as Bool)` to include all of the original bindings (e.g. the renaming is strictly additive). It's important to specify quotes so that your shell passes the package name and thinning/renaming list as a single argument to GHC.

Package imports with thinning/renaming do not hide other versions of the package: e.g. if containers-0.9 is already exposed, `-package "containers-0.8 (Data.List as ListV8)"` will only add an additional binding to the environment. Similarly, `-package "base (Data.Bool as Bool)" -package "base (Data.List as List)"` is equivalent to `-package "base (Data.Bool as Bool, Data.List as List)"`. Literal names must refer to modules defined by the original package, so for example `-package "base (Data.Bool as Bool, Bool as Baz)"` is invalid unless there was a `Bool` module defined in the original package. Hiding a package also clears all of its renamings.

[1] it used to in GHC 6.4, but not since 6.6

You can use renaming to provide an alternate prelude, e.g. `-hide-all-packages -package "basic-prelude (BasicPrelude as Prelude)"`, in lieu of the *Rebindable syntax and the implicit Prelude import* (page 295) extension.

5.9.5 Package Databases

A package database is where the details about installed packages are stored. It is a directory, usually called `package.conf.d`, that contains a file for each package, together with a binary cache of the package data in the file `package.cache`. Normally you won't need to look at or modify the contents of a package database directly; all management of package databases can be done through the **ghc-pkg** tool (see *Package management (the ghc-pkg command)* (page 228)).

GHC knows about two package databases in particular:

- The *global package database*, which comes with your GHC installation, e.g. `/usr/lib/ghc-6.12.1/package.conf.d`.
- The *user package database* private to each user. On Unix systems this will be `$XDG_DATA_HOME/ghc/arch-os-version/package.conf.d`, and on Windows it will be something like `C:\Documents And Settings\user\ghc\package.conf.d`. The `ghc-pkg` tool knows where this file should be located, and will create it if it doesn't exist (see *Package management (the ghc-pkg command)* (page 228)).

Package database stack: Package databases are arranged in a stack structure. When GHC starts up it adds the global and the user package databases to the stack, in that order, unless *GHC_PACKAGE_PATH* (page 225) is specified. When *GHC_PACKAGE_PATH* is specified then it will determine the initial database stack. Several command line options described below can further manipulate this initial stack. You can see GHC's effective package database stack by running GHC with the *-v* (page 76) flag.

This stack structure means that the order of *-package-db ⟨file⟩* (page 224) flags or *GHC_PACKAGE_PATH* (page 225) is important. Each substack of the stack must be well formed (packages in databases on top of the stack can refer to packages below, but not vice versa).

Package shadowing: When multiple package databases are in use it is possible, though rarely, that the same installed package id is present in more than one database. In that case, packages closer to the top of the stack will override (*shadow*) those below them. If the conflicting packages are found to be equivalent (by ABI hash comparison) then one of them replaces all references to the other, otherwise the overridden package and all those depending on it will be removed.

Package version selection: When selecting a package, GHC will search for packages in all available databases. If multiple versions of the same package are available the latest non-broken version will be chosen.

Version conflict resolution: If multiple instances of a package version chosen by GHC are available then GHC will choose an unspecified instance.

You can control GHC's package database stack using the following options:

-package-db ⟨file⟩
  Add the package database ⟨file⟩ on top of the current stack.

-no-global-package-db
  Remove the global package database from the package database stack.

-no-user-package-db
  Prevent loading of the user's local package database in the initial stack.

-clear-package-db
Reset the current package database stack. This option removes every previously specified package database (including those read from the *GHC_PACKAGE_PATH* (page 225) environment variable) from the package database stack.

-global-package-db
Add the global package database on top of the current stack. This option can be used after *-no-global-package-db* (page 224) to specify the position in the stack where the global package database should be loaded.

-user-package-db
Add the user's package database on top of the current stack. This option can be used after *-no-user-package-db* (page 224) to specify the position in the stack where the user's package database should be loaded.

5.9.5.1 The GHC_PACKAGE_PATH environment variable

GHC_PACKAGE_PATH
The *GHC_PACKAGE_PATH* environment variable may be set to a :-separated (;-separated on Windows) list of files containing package databases. This list of package databases, used by GHC and ghc-pkg, specifies a stack of package databases from top to bottom. This order was chosen to match the behaviour of the `PATH` environment variable where entries earlier in the PATH override ones that come later. See *Package Databases* (page 224) for details on how the package database stack is used.

Normally *GHC_PACKAGE_PATH* replaces the default package stack. For example, all of the following commands are equivalent, creating a stack with db1 at the top followed by db2 (use `;` instead of `:` on Windows):

```
$ ghc -clear-package-db -package-db db2.conf -package-db db1.conf
$ env GHC_PACKAGE_PATH=db1.conf:db2.conf ghc
$ env GHC_PACKAGE_PATH=db2.conf ghc -package-db db1.conf
```

However, if *GHC_PACKAGE_PATH* ends in a separator, the default databases (i.e. the user and global package databases, in that order) are appended to the path. For example, to augment the usual set of packages with a database of your own, you could say (on Unix):

```
$ export GHC_PACKAGE_PATH=$XDG_DATA_HOME/.my-ghc-packages.conf:
```

To check whether your *GHC_PACKAGE_PATH* setting is doing the right thing, `ghc-pkg list` will list all the databases in use, in the reverse order they are searched.

5.9.5.2 Package environments

A *package environment file* is a file that tells `ghc` precisely which packages should be visible. It can be used to create environments for `ghc` or `ghci` that are local to a shell session or to some file system location. They are intended to be managed by build/package tools, to enable `ghc` and `ghci` to automatically use an environment created by the tool.

In the case of `ghci`, the environment file will be read once, during initialisation. If the file changes then you have to restart GHCi to reflect the updated file.

The file contains package IDs and optionally package databases, one directive per line:

```
clear-package-db
global-package-db
user-package-db
package-db db.d/
package-id id_1
package-id id_2
...
package-id id_n
```

If such a package environment is found, it is equivalent to passing these command line arguments to ghc:

```
-hide-all-packages
-clear-package-db
-global-package-db
-user-package-db
-package-db db.d/
-package-id id_1
-package-id id_2
...
-package-id id_n
```

Note the implicit *-hide-all-packages* (page 221) and the fact that it is *-package-id ⟨unit-id⟩* (page 221), not *-package ⟨pkg⟩* (page 221). This is because the environment specifies precisely which packages should be visible.

Note that for the `package-db` directive, if a relative path is given it must be relative to the location of the package environment file.

-package-env ⟨file⟩|⟨name⟩
: Use the package environment in ⟨file⟩, or in `$XDG_DATA_HOME/ghc/arch-os-version/environments/⟨name⟩` If set to `-` no package environment is read.

GHC_ENVIRONMENT
: Specifies the path to the package environment file to be used by GHC. Overridden by the *-package-env ⟨file⟩|⟨name⟩* (page 226) flag if set.

In order, `ghc` will look for the package environment in the following locations:

- File ⟨file⟩ if you pass the option *-package-env ⟨file⟩|⟨name⟩* (page 226).
- File `$XDG_DATA_HOME/ghc/arch-os-version/environments/name` if you pass the option `-package-env ⟨name⟩`.
- File ⟨file⟩ if the environment variable *GHC_ENVIRONMENT* (page 226) is set to ⟨file⟩.
- File `$XDG_DATA_HOME/ghc/arch-os-version/environments/name` if the environment variable *GHC_ENVIRONMENT* (page 226) is set to ⟨name⟩.

Additionally, unless `-hide-all-packages` is specified `ghc` will also look for the package environment in the following locations:

- File `.ghc.environment.arch-os-version` if it exists in the current directory or any parent directory (but not the user's home directory).
- File `$XDG_DATA_HOME/ghc/arch-os-version/environments/default` if it exists.

Package environments can be modified by further command line arguments; for example, if you specify `-package foo` on the command line, then package ⟨foo⟩ will be visible even if it's not listed in the currently active package environment.

5.9.6 Installed package IDs, dependencies, and broken packages

Each installed package has a unique identifier (the "installed package ID"), which distinguishes it from all other installed packages on the system. To see the installed package IDs associated with each installed package, use `ghc-pkg list -v`:

```
$ ghc-pkg list -v
using cache: /usr/lib/ghc-6.12.1/package.conf.d/package.cache
/usr/lib/ghc-6.12.1/package.conf.d
   Cabal-1.7.4 (Cabal-1.7.4-48f5247e06853af93593883240e11238)
   array-0.2.0.1 (array-0.2.0.1-9cbf76a576b6ee9c1f880cf171a0928d)
   base-3.0.3.0 (base-3.0.3.0-6cbb157b9ae852096266e113b8fac4a2)
   base-4.2.0.0 (base-4.2.0.0-247bb20cde37c3ef4093ee124e04bc1c)
   ...
```

The string in parentheses after the package name is the installed package ID: it normally begins with the package name and version, and ends in a hash string derived from the compiled package. Dependencies between packages are expressed in terms of installed package IDs, rather than just packages and versions. For example, take a look at the dependencies of the `haskell98` package:

```
$ ghc-pkg field haskell98 depends
depends: array-0.2.0.1-9cbf76a576b6ee9c1f880cf171a0928d
         base-4.2.0.0-247bb20cde37c3ef4093ee124e04bc1c
         directory-1.0.0.2-f51711bc872c35ce4a453aa19c799008
         old-locale-1.0.0.1-d17c9777c8ee53a0d459734e27f2b8e9
         old-time-1.0.0.1-1c0d8ea38056e5087ef1e75cb0d139d1
         process-1.0.1.1-d8fc6d3baf44678a29b9d59ca0ad5780
         random-1.0.0.1-423d08c90f004795fd10e60384ce6561
```

The purpose of the installed package ID is to detect problems caused by re-installing a package without also recompiling the packages that depend on it. Recompiling dependencies is necessary, because the newly compiled package may have a different ABI (Application Binary Interface) than the previous version, even if both packages were built from the same source code using the same compiler. With installed package IDs, a recompiled package will have a different installed package ID from the previous version, so packages that depended on the previous version are now orphaned - one of their dependencies is not satisfied. Packages that are broken in this way are shown in the `ghc-pkg list` output either in red (if possible) or otherwise surrounded by braces. In the following example, we have recompiled and reinstalled the `filepath` package, and this has caused various dependencies including `Cabal` to break:

```
$ ghc-pkg list
WARNING: there are broken packages.  Run 'ghc-pkg check' for more details.
/usr/lib/ghc-6.12.1/package.conf.d:
    {Cabal-1.7.4}
    array-0.2.0.1
    base-3.0.3.0
    ... etc ...
```

Additionally, `ghc-pkg list` reminds you that there are broken packages and suggests `ghc-pkg check`, which displays more information about the nature of the failure:

```
$ ghc-pkg check
There are problems in package ghc-6.12.1:
  dependency "filepath-1.1.0.1-87511764eb0af2bce4db05e702750e63" doesn't exist
There are problems in package haskeline-0.6.2:
```

(continues on next page)

(continued from previous page)

```
  dependency "filepath-1.1.0.1-87511764eb0af2bce4db05e702750e63" doesn't exist
There are problems in package Cabal-1.7.4:
  dependency "filepath-1.1.0.1-87511764eb0af2bce4db05e702750e63" doesn't exist
There are problems in package process-1.0.1.1:
  dependency "filepath-1.1.0.1-87511764eb0af2bce4db05e702750e63" doesn't exist
There are problems in package directory-1.0.0.2:
  dependency "filepath-1.1.0.1-87511764eb0af2bce4db05e702750e63" doesn't exist

The following packages are broken, either because they have a problem
listed above, or because they depend on a broken package.
ghc-6.12.1
haskeline-0.6.2
Cabal-1.7.4
process-1.0.1.1
directory-1.0.0.2
bin-package-db-0.0.0.0
hpc-0.5.0.2
haskell98-1.0.1.0
```

To fix the problem, you need to recompile the broken packages against the new dependencies. The easiest way to do this is to use `cabal-install`, or download the packages from HackageDB and build and install them as normal.

Be careful not to recompile any packages that GHC itself depends on, as this may render the `ghc` package itself broken, and `ghc` cannot be simply recompiled. The only way to recover from this would be to re-install GHC.

5.9.7 Package management (the `ghc-pkg` command)

The **ghc-pkg** tool is for querying and modifying package databases. To see what package databases are in use, use `ghc-pkg list`. The stack of databases that **ghc-pkg** knows about can be modified using the *GHC_PACKAGE_PATH* (page 225) environment variable (see *The GHC_PACKAGE_PATH environment variable* (page 225), and using *-package-db ⟨file⟩* (page 224) options on the **ghc-pkg** command line.

When asked to modify a database, `ghc-pkg` modifies the global database by default. Specifying `--user` causes it to act on the user database, or `--package-db` can be used to act on another database entirely. When multiple of these options are given, the rightmost one is used as the database to act upon.

Commands that query the package database (list, latest, describe, field, dot) operate on the list of databases specified by the flags `--user`, `--global`, and `--package-db`. If none of these flags are given, the default is `--global --user`.

If the environment variable *GHC_PACKAGE_PATH* (page 225) is set, and its value does not end in a separator (`:` on Unix, `;` on Windows), then the last database is considered to be the global database, and will be modified by default by `ghc-pkg`. The intention here is that `GHC_PACKAGE_PATH` can be used to create a virtual package environment into which Cabal packages can be installed without setting anything other than `GHC_PACKAGE_PATH`.

The `ghc-pkg` program may be run in the ways listed below. Where a package name is required, the package can be named in full including the version number (e.g. `network-1.0`), or without the version number. Naming a package without the version number matches all versions of the package; the specified action will be applied to all the matching packages. A package specifier that matches all version of the package can also be written `⟨pkg⟩ -*`, to make it clearer that multiple packages are being matched. To match against the installed package ID instead of just package name and version, pass the `--ipid` flag.

`ghc-pkg init path` Creates a new, empty, package database at ⟨path⟩, which must not already exist.

`ghc-pkg register ⟨file⟩` Reads a package specification from ⟨file⟩ (which may be "`-`" to indicate standard input), and adds it to the database of installed packages. The syntax of ⟨file⟩ is given in *InstalledPackageInfo: a package specification* (page 232).

The package specification must be a package that isn't already installed.

`ghc-pkg update ⟨file⟩` The same as `register`, except that if a package of the same name is already installed, it is replaced by the new one.

`ghc-pkg unregister ⟨P⟩` Remove the specified package from the database.

`ghc-pkg check` Check consistency of dependencies in the package database, and report packages that have missing dependencies.

`ghc-pkg expose ⟨P⟩` Sets the `exposed` flag for package ⟨P⟩ to `True`.

`ghc-pkg hide ⟨P⟩` Sets the `exposed` flag for package ⟨P⟩ to `False`.

`ghc-pkg trust ⟨P⟩` Sets the `trusted` flag for package ⟨P⟩ to `True`.

`ghc-pkg distrust ⟨P⟩` Sets the `trusted` flag for package ⟨P⟩ to `False`.

`ghc-pkg list [⟨P⟩] [--simple-output]` This option displays the currently installed packages, for each of the databases known to `ghc-pkg`. That includes the global database, the user's local database, and any further files specified using the `-f` option on the command line.

Hidden packages (those for which the `exposed` flag is `False`) are shown in parentheses in the list of packages.

If an optional package identifier ⟨P⟩ is given, then only packages matching that identifier are shown.

If the option `--simple-output` is given, then the packages are listed on a single line separated by spaces, and the database names are not included. This is intended to make it easier to parse the output of `ghc-pkg list` using a script.

`ghc-pkg find-module ⟨M⟩ [--simple-output]` This option lists registered packages exposing module ⟨M⟩. Examples:

```
$ ghc-pkg find-module Var
c:/fptools/validate/ghc/driver/package.conf.inplace:
    (ghc-6.9.20080428)

$ ghc-pkg find-module Data.Sequence
c:/fptools/validate/ghc/driver/package.conf.inplace:
    containers-0.1
```

Otherwise, it behaves like `ghc-pkg list`, including options.

`ghc-pkg latest ⟨P⟩` Prints the latest available version of package ⟨P⟩.

`ghc-pkg describe ⟨P⟩` Emit the full description of the specified package. The description is in the form of an `InstalledPackageInfo`, the same as the input file format for `ghc-pkg register`. See *InstalledPackageInfo: a package specification* (page 232) for details.

If the pattern matches multiple packages, the description for each package is emitted, separated by the string `---` on a line by itself.

ghc-pkg field ⟨P⟩ ⟨field⟩[,⟨field⟩]* Show just a single field of the installed package description for P. Multiple fields can be selected by separating them with commas

ghc-pkg dot Generate a graph of the package dependencies in a form suitable for input for the graphviz tools. For example, to generate a PDF of the dependency graph:

```
ghc-pkg dot | tred | dot -Tpdf >pkgs.pdf
```

ghc-pkg dump Emit the full description of every package, in the form of an `InstalledPackageInfo`. Multiple package descriptions are separated by the string `---` on a line by itself.

This is almost the same as `ghc-pkg describe '*'`, except that `ghc-pkg dump` is intended for use by tools that parse the results, so for example where `ghc-pkg describe '*'` will emit an error if it can't find any packages that match the pattern, `ghc-pkg dump` will simply emit nothing.

ghc-pkg recache Re-creates the binary cache file `package.cache` for the selected database. This may be necessary if the cache has somehow become out-of-sync with the contents of the database (`ghc-pkg` will warn you if this might be the case).

The other time when `ghc-pkg recache` is useful is for registering packages manually: it is possible to register a package by simply putting the appropriate file in the package database directory and invoking `ghc-pkg recache` to update the cache. This method of registering packages may be more convenient for automated packaging systems.

Substring matching is supported for ⟨M⟩ in `find-module` and for ⟨P⟩ in `list`, `describe`, and `field`, where a `'*'` indicates open substring ends (`prefix*`, `*suffix`, `*infix*`). Examples (output omitted):

```
-- list all regex-related packages
ghc-pkg list '*regex*' --ignore-case
-- list all string-related packages
ghc-pkg list '*string*' --ignore-case
-- list OpenGL-related packages
ghc-pkg list '*gl*' --ignore-case
-- list packages exporting modules in the Data hierarchy
ghc-pkg find-module 'Data.*'
-- list packages exporting Monad modules
ghc-pkg find-module '*Monad*'
-- list names and maintainers for all packages
ghc-pkg field '*' name,maintainer
-- list location of haddock htmls for all packages
ghc-pkg field '*' haddock-html
-- dump the whole database
ghc-pkg describe '*'
```

Additionally, the following flags are accepted by `ghc-pkg`:

-f ⟨file⟩, -package-db ⟨file⟩ Adds ⟨file⟩ to the stack of package databases. Additionally, ⟨file⟩ will also be the database modified by a `register`, `unregister`, `expose` or `hide` command, unless it is overridden by a later `--package-db`, `--user` or `--global` option.

--force Causes `ghc-pkg` to ignore missing dependencies, directories and libraries when registering a package, and just go ahead and add it anyway. This might be useful if your package installation system needs to add the package to GHC before building and installing the files.

--global Operate on the global package database (this is the default). This flag affects the `register`, `update`, `unregister`, `expose`, and `hide` commands.

--help, -? Outputs the command-line syntax.

--user Operate on the current user's local package database. This flag affects the `register`, `update`, `unregister`, `expose`, and `hide` commands.

-v [⟨n⟩], --verbose [=⟨n⟩] Control verbosity. Verbosity levels range from 0-2, where the default is 1, and `-v` alone selects level 2.

-V; --version Output the `ghc-pkg` version number.

--ipid, --unit-id Causes `ghc-pkg` to interpret arguments as installed unit IDs (e.g., an identifier like `unix-2.3.1.0-de7803f1a8cd88d2161b29b083c94240`). This is useful if providing just the package name and version are ambiguous (in old versions of GHC, this was guaranteed to be unique, but this invariant no longer necessarily holds).

5.9.8 Building a package from Haskell source

We don't recommend building packages the hard way. Instead, use the Cabal infrastructure if possible. If your package is particularly complicated or requires a lot of configuration, then you might have to fall back to the low-level mechanisms, so a few hints for those brave souls follow.

You need to build an "installed package info" file for passing to `ghc-pkg` when installing your package. The contents of this file are described in *InstalledPackageInfo: a package specification* (page 232).

The Haskell code in a package may be built into one or more archive libraries (e.g. `libHSfoo.a`), or a single shared object (e.g. `libHSfoo.dll/.so/.dylib`). The restriction to a single shared object is because the package system is used to tell the compiler when it should make an inter-shared-object call rather than an intra-shared-object-call call (inter-shared-object calls require an extra indirection).

- Building a static library is done by using the **ar** tool, like so:

  ```
  ar cqs libHSfoo-1.0.a A.o B.o C.o ...
  ```

  where `A.o`, `B.o` and so on are the compiled Haskell modules, and `libHSfoo.a` is the library you wish to create. The syntax may differ slightly on your system, so check the documentation if you run into difficulties.

- To load a package `foo`, GHCi can load its `libHSfoo.a` library directly, but it can also load a package in the form of a single `HSfoo.o` file that has been pre-linked. Loading the `.o` file is slightly quicker, but at the expense of having another copy of the compiled package. The rule of thumb is that if the modules of the package were compiled with *`-split-sections`* (page 246) then building the `HSfoo.o` is worthwhile because it saves time when loading the package into GHCi. Without *`-split-sections`* (page 246), there is not much difference in load time between the `.o` and `.a` libraries, so it is better to save the disk space and only keep the `.a` around. In a GHC distribution we provide `.o` files for most packages except the GHC package itself.

  The `HSfoo.o` file is built by Cabal automatically; use `--disable-library-for-ghci` to disable it. To build one manually, the following GNU `ld` command can be used:

  ```
  ld -r --whole-archive -o HSfoo.o libHSfoo.a
  ```

  (replace `--whole-archive` with `-all_load` on MacOS X)

- When building the package as shared library, GHC can be used to perform the link step. This hides some of the details out the underlying linker and provides a common interface to all shared object variants that are supported by GHC (DLLs, ELF DSOs, and Mac OS dylibs). The shared object must be named in specific way for two reasons: (1) the name must contain the GHC compiler version, so that two library variants don't collide that are compiled by different versions of GHC and that therefore are most likely incompatible with respect to calling conventions, (2) it must be different from the static name otherwise we would not be able to control the linker as precisely as necessary to make the *-static* (page 246)/*-dynamic* (page 246) flags work, see *Options affecting linking* (page 245).

```
ghc -shared -dynamic -o libHSfoo-1.0-ghcGHCVersion.so A.o B.o C.o
```

Using GHC's version number in the shared object name allows different library versions compiled by different GHC versions to be installed in standard system locations, e.g. under *nix `/usr/lib`. To obtain the version number of GHC invoke `ghc --numeric-version` and use its output in place of ⟨GHCVersion⟩. See also *Options affecting code generation* (page 243) on how object files must be prepared for shared object linking.

- When building a shared library, care must be taken to ensure that the resulting object is named appropriately. In particular, GHC expects the name of a shared object to have the form `libHS<unit id>-ghc<ghc version>.<ext>` where *unit id* is the unit ID given during compilation via the *-this-unit-id ⟨unit-id⟩* (page 222) flag, *ghc version* is the version of GHC that produced/consumes the object and *ext* is the host system's usual file extension for shared objects.

To compile a module which is to be part of a new package, use the `-package-name` (to identify the name of the package) and `-library-name` (to identify the version and the version hashes of its identities.) options (*Using Packages* (page 219)). Failure to use these options when compiling a package will probably result in disaster, but you will only discover later when you attempt to import modules from the package. At this point GHC will complain that the package name it was expecting the module to come from is not the same as the package name stored in the `.hi` file.

It is worth noting with shared objects, when each package is built as a single shared object file, since a reference to a shared object costs an extra indirection, intra-package references are cheaper than inter-package references. Of course, this applies to the `main` package as well.

5.9.9 `InstalledPackageInfo`: a package specification

A package specification is a Haskell record; in particular, it is the record Distribution.InstalledPackageInfo.InstalledPackageInfo in the module Distribution.InstalledPackageInfo, which is part of the Cabal package distributed with GHC.

An `InstalledPackageInfo` has a human readable/writable syntax. The functions `parseInstalledPackageInfo` and `showInstalledPackageInfo` read and write this syntax respectively. Here's an example of the `InstalledPackageInfo` for the `unix` package:

```
$ ghc-pkg describe unix
name: unix
version: 2.3.1.0
id: unix-2.3.1.0-de7803f1a8cd88d2161b29b083c94240
license: BSD3
copyright:
maintainer: libraries@haskell.org
stability:
```

(continues on next page)

(continued from previous page)

```
homepage:
package-url:
description: This package gives you access to the set of operating system
             services standardised by POSIX 1003.1b (or the IEEE Portable
             Operating System Interface for Computing Environments -
             IEEE Std. 1003.1).
             .
             The package is not supported under Windows (except under Cygwin).
category: System
author:
exposed: True
exposed-modules: System.Posix System.Posix.DynamicLinker.Module
                 System.Posix.DynamicLinker.Prim System.Posix.Directory
                 System.Posix.DynamicLinker System.Posix.Env System.Posix.Error
                 System.Posix.Files System.Posix.IO System.Posix.Process
                 System.Posix.Process.Internals System.Posix.Resource
                 System.Posix.Temp System.Posix.Terminal System.Posix.Time
                 System.Posix.Unistd System.Posix.User System.Posix.Signals
                 System.Posix.Signals.Exts System.Posix.Semaphore
                 System.Posix.SharedMem
hidden-modules:
trusted: False
import-dirs: /usr/lib/ghc-6.12.1/unix-2.3.1.0
library-dirs: /usr/lib/ghc-6.12.1/unix-2.3.1.0
hs-libraries: HSunix-2.3.1.0
extra-libraries: rt util dl
extra-ghci-libraries:
include-dirs: /usr/lib/ghc-6.12.1/unix-2.3.1.0/include
includes: HsUnix.h execvpe.h
depends: base-4.2.0.0-247bb20cde37c3ef4093ee124e04bc1c
hugs-options:
cc-options:
ld-options:
framework-dirs:
frameworks:
haddock-interfaces: /usr/share/doc/ghc/html/libraries/unix/unix.haddock
haddock-html: /usr/share/doc/ghc/html/libraries/unix
```

Here is a brief description of the syntax of this file:

A package description consists of a number of field/value pairs. A field starts with the field name in the left-hand column followed by a "`:`", and the value continues until the next line that begins in the left-hand column, or the end of file.

The syntax of the value depends on the field. The various field types are:

freeform Any arbitrary string, no interpretation or parsing is done.

string A sequence of non-space characters, or a sequence of arbitrary characters surrounded by quotes `"...."`.

string list A sequence of strings, separated by commas. The sequence may be empty.

In addition, there are some fields with special syntax (e.g. package names, version, dependencies).

The allowed fields, with their types, are:

`name` (string) The package's name (without the version).

id (string) The installed package ID. It is up to you to choose a suitable one.

version (string) The package's version, usually in the form `A.B` (any number of components are allowed).

license (string) The type of license under which this package is distributed. This field is a value of the Distribution.License.License type.

license-file (optional string) The name of a file giving detailed license information for this package.

copyright (optional freeform) The copyright string.

maintainer (optional freeform) The email address of the package's maintainer.

stability (optional freeform) A string describing the stability of the package (e.g. stable, provisional or experimental).

homepage (optional freeform) URL of the package's home page.

package-url (optional freeform) URL of a downloadable distribution for this package. The distribution should be a Cabal package.

description (optional freeform) Description of the package.

category (optional freeform) Which category the package belongs to. This field is for use in conjunction with a future centralised package distribution framework, tentatively titled Hackage.

author (optional freeform) Author of the package.

exposed (bool) Whether the package is exposed or not.

exposed-modules (string list) modules exposed by this package.

hidden-modules (string list) modules provided by this package, but not exposed to the programmer. These modules cannot be imported, but they are still subject to the overlapping constraint: no other package in the same program may provide a module of the same name.

reexported-modules Modules reexported by this package. This list takes the form of `pkg:OldName as NewName (A@orig-pkg-0.1-HASH)`: the first portion of the string is the user-written reexport specification (possibly omitting the package qualifier and the renaming), while the parenthetical is the original package which exposed the module under are particular name. Reexported modules have a relaxed overlap constraint: it's permissible for two packages to reexport the same module as the same name if the reexported moduleis identical.

trusted (bool) Whether the package is trusted or not.

import-dirs (string list) A list of directories containing interface files (`.hi` files) for this package.

If the package contains profiling libraries, then the interface files for those library modules should have the suffix `.p_hi`. So the package can contain both normal and profiling versions of the same library without conflict (see also `library_dirs` below).

library-dirs (string list) A list of directories containing libraries for this package.

hs-libraries (string list) A list of libraries containing Haskell code for this package, with the `.a` or `.dll` suffix omitted. When packages are built as libraries, the `lib` prefix is also omitted.

For use with GHCi, each library should have an object file too. The name of the object file does *not* have a `lib` prefix, and has the normal object suffix for your platform.

For example, if we specify a Haskell library as `HSfoo` in the package spec, then the various flavours of library that GHC actually uses will be called:

`libHSfoo.a` The name of the library on Unix and Windows (mingw) systems.

`HSfoo.dll` The name of the dynamic library on Windows systems (optional).

`HSfoo.o`; `HSfoo.obj` The object version of the library used by GHCi.

`extra-libraries` (string list) A list of extra libraries for this package. The difference between `hs-libraries` and `extra-libraries` is that `hs-libraries` normally have several versions, to support profiling, parallel and other build options. The various versions are given different suffixes to distinguish them, for example the profiling version of the standard prelude library is named `libHSbase_p.a`, with the `_p` indicating that this is a profiling version. The suffix is added automatically by GHC for `hs-libraries` only, no suffix is added for libraries in `extra-libraries`.

The libraries listed in `extra-libraries` may be any libraries supported by your system's linker, including dynamic libraries (`.so` on Unix, `.DLL` on Windows).

Also, `extra-libraries` are placed on the linker command line after the `hs-libraries` for the same package. If your package has dependencies in the other direction (i.e. `extra-libraries` depends on `hs-libraries`), and the libraries are static, you might need to make two separate packages.

`include-dirs` (string list) A list of directories containing C includes for this package.

`includes` (string list) A list of files to include for via-C compilations using this package. Typically the include file(s) will contain function prototypes for any C functions used in the package, in case they end up being called as a result of Haskell functions from the package being inlined.

`depends` (package id list) Packages on which this package depends.

`hugs-options` (string list) Options to pass to Hugs for this package.

`cc-options` (string list) Extra arguments to be added to the gcc command line when this package is being used (only for via-C compilations).

`ld-options` (string list) Extra arguments to be added to the `gcc` command line (for linking) when this package is being used.

`framework-dirs` (string list) On Darwin/MacOS X, a list of directories containing frameworks for this package. This corresponds to the `-framework-path` option. It is ignored on all other platforms.

`frameworks` (string list) On Darwin/MacOS X, a list of frameworks to link to. This corresponds to the `-framework` option. Take a look at Apple's developer documentation to find out what frameworks actually are. This entry is ignored on all other platforms.

`haddock-interfaces` (string list) A list of filenames containing Haddock interface files (`.haddock` files) for this package.

`haddock-html` (optional string) The directory containing the Haddock-generated HTML for this package.

5.9.10 Linking against C++ libraries

Use of C++ libraries requires that the user link against the host system's C++ standard library. As the configuration necessary to achieve this is generally quite platform-dependent, GHC provides a built-in package, `system-cxx-std-lib`. This package captures the configuration necessary for linking against the C++ standard library and can be used via the *-package ⟨pkg⟩* (page 221) flag or the Cabal `build-depends` field to link code against the C++ standard library.

5.10 GHC Backends

GHC supports multiple backend code generators. This is the part of the compiler responsible for taking the last intermediate representation that GHC uses (a form called Cmm that is a simple, C like language) and compiling it to executable code. The backends that GHC support are described below.

5.10.1 Native Code Generator (`-fasm`)

The default backend for GHC. It is a native code generator, compiling Cmm all the way to assembly code. It is the fastest backend and generally produces good performance code. It has the best support for compiling shared libraries. Select it with the `-fasm` flag.

5.10.2 LLVM Code Generator (`-fllvm`)

This is an alternative backend that uses the LLVM compiler to produce executable code. It generally produces code with performance as good as the native code generator but for some cases can produce much faster code. This is especially true for numeric, array heavy code using packages like vector. The penalty is a significant increase in compilation times. Select the LLVM backend with the *-fllvm* (page 243) flag.

You must install and have LLVM available on your `PATH` for the LLVM code generator to work. Specifically GHC needs to be able to call the `opt` and `llc` tools. Secondly, if you are running Mac OS X with LLVM 3.0 or greater then you also need the Clang C compiler compiler available on your `PATH`.

Note: Note that this GHC release expects an LLVM version in the 13 up to 16 (not inclusive) release series.

To install LLVM and Clang:

- *Linux*: Use your package management tool.
- *Mac OS X*: Clang is included by default on recent OS X machines when Xcode is installed (from 10.6 and later). LLVM is not included. In order to use the LLVM based code generator, you should install the Homebrew package manager for OS X. Alternatively you can download binaries for LLVM and Clang from here.
- *Windows*: You should download binaries for LLVM and clang from here.

5.10.3 C Code Generator (-fvia-C)

-fvia-C

Use the C code generator. Only supposed in unregisterised GHC builds.

This is the oldest code generator in GHC and is generally not included any more having been deprecated around GHC 7.0. Select it with the *-fvia-C* (page 237) flag.

The C code generator is only supported when GHC is built in unregisterised mode, a mode where GHC produces "portable" C code as output to facilitate porting GHC itself to a new platform. This mode produces much slower code though so it's unlikely your version of GHC was built this way. If it has then the native code generator probably won't be available. You can check this information by calling `ghc --info` (see *--info* (page 69)).

5.10.4 JavaScript Code Generator

This is an alternative code generator included in GHC 9.6 and above. It generates ECMA-262 compliant JavaScript and is included as a technical preview. At time of writing, it is being actively developed but is not suitable for serious projects and production environments. The JavaScript backend is not distributed in the GHC bindist and requires a manual build. See building the JavaScript backend page on the GHC wiki for build instructions.

A JavaScript cross-compiling GHC produces an executable script, and a directory of the same name suffixed with `.jsexe`. For example, compiling a file named `Foo.hs` will produce an executable script `Foo` and a `Foo.jsexe` directory. The script is a thin wrapper that calls Node.js on the payload of the compiled Haskell code and can be run in the usual way, e.g., `./Foo`, as long as `node` is in your environment . The actual payload is in `<ModuleName>.jsexe/all.js`, for example `Foo.jsexe/all.js`. This file is the Haskell program cross-compiled to JavaScript *concrete syntax* and can be wrapped in a `<script>` HTML tag. For a breakdown of the rest of the build artifacts see the compiler output section in the wiki.

5.10.5 Unregisterised compilation

The term "unregisterised" really means "compile via vanilla C", disabling some of the platform-specific tricks that GHC normally uses to make programs go faster. When compiling unregisterised, GHC simply generates a C file which is compiled via gcc.

When GHC is built in unregisterised mode only the C code generator is available. Neither the LLVM nor native code generator can be used by an unregisterised build.

Unregisterised compilation can be useful when porting GHC to a new machine, since it reduces the prerequisite tools to `gcc`, `as`, and `ld` and nothing more, and furthermore the amount of platform-specific code that needs to be written in order to get unregisterised compilation going is usually fairly small.

Unregisterised compilation cannot be selected at compile-time; you have to build GHC with the appropriate options set. Consult the GHC Building Guide for details.

You can check if your GHC is unregisterised by calling `ghc --print-unregisterised` (see *--print-unregisterised* (page 71)) or `ghc --info` (see *--info* (page 69)).

5.11 Options related to a particular phase

5.11.1 Replacing the program for one or more phases

You may specify that a different program be used for one of the phases of the compilation system, in place of whatever the `ghc` has wired into it. For example, you might want to try a different assembler. The following options allow you to change the external program used for a given compilation phase:

-pgmL ⟨cmd⟩
    Use ⟨cmd⟩ as the literate pre-processor.

-pgmP ⟨cmd⟩
    Use ⟨cmd⟩ as the C pre-processor (with *-cpp* (page 240) only).

-pgmc ⟨cmd⟩
    Use ⟨cmd⟩ as the C compiler.

-pgmcxx ⟨cmd⟩
    Use ⟨cmd⟩ as the C++ compiler.

-pgmlo ⟨cmd⟩
    Use ⟨cmd⟩ as the LLVM optimiser.

-pgmlc ⟨cmd⟩
    Use ⟨cmd⟩ as the LLVM compiler.

-pgmlas ⟨cmd⟩
    Use ⟨cmd⟩ as the LLVM assembler

-pgms ⟨cmd⟩
    Use ⟨cmd⟩ as the splitter.

-pgma ⟨cmd⟩
    Use ⟨cmd⟩ as the assembler.

-pgml ⟨cmd⟩
    Use ⟨cmd⟩ as the linker.

-pgmlm ⟨cmd⟩
    Use ⟨cmd⟩ as the linker when merging object files (e.g. when generating joined objects for loading into GHCi).

-pgmF ⟨cmd⟩
    Use ⟨cmd⟩ as the pre-processor (with *-F* (page 243) only).

-pgmotool ⟨cmd⟩
    Use ⟨cmd⟩ as the program to inspect mach-o dynamic libraries and executables to read the dynamic library dependencies. We will compute the necessary `runpath``s to embed for the dependencies based on the result of the ``otool` call.

-pgminstall_name_tool ⟨cmd⟩
    Use ⟨cmd⟩ as the program to inject `runpath``s into mach-o dynamic libraries and executables. As detected by the ``otool` call.

-pgmwindres ⟨cmd⟩
    Use ⟨cmd⟩ as the program to use for embedding manifests on Windows. Normally this is the program `windres`, which is supplied with a GHC installation. See `-fno-embed-manifest` in *Options affecting linking* (page 245).

-pgmi ⟨cmd⟩
Use ⟨cmd⟩ as the external interpreter command (see *Running the interpreter in a separate process* (page 60)). Default: `ghc-iserv-prof` if *-prof* (page 656) is enabled, `ghc-iserv-dyn` if *-dynamic* (page 246) is enabled, or `ghc-iserv` otherwise.

5.11.2 Forcing options to a particular phase

Options can be forced through to a particular compilation phase, using the following flags:

-optL ⟨option⟩
Pass ⟨option⟩ to the literate pre-processor

-optP ⟨option⟩
Pass ⟨option⟩ to CPP (makes sense only if *-cpp* (page 240) is also on).

-optF ⟨option⟩
Pass ⟨option⟩ to the custom pre-processor (see *Options affecting a Haskell pre-processor* (page 243)).

-optc ⟨option⟩
Pass ⟨option⟩ to the C compiler.

-pgmc-supports-no-pie
Does the same thing as `-pgml-supports-no-pie`, which replaced it.

-pgml-supports-no-pie
When `-pgml` is used, GHC by default will never pass the `-no-pie` command line flag. The rationale is that it is not known whether the specified compiler used for linking (recall we use a C compiler to invoke the linker on our behalf) will support it. This flag can be used to indicate that `-no-pie` is supported. It has to be passed after `-pgml`.

This flag is not necessary when `-pgmc` is not used, since GHC remembers whether the default C compiler supports `-no-pie` in an internal settings file.

-optcxx ⟨option⟩
Pass ⟨option⟩ to the C++ compiler.

-optlo ⟨option⟩
Pass ⟨option⟩ to the LLVM optimiser.

-optlc ⟨option⟩
Pass ⟨option⟩ to the LLVM compiler.

-optlas ⟨option⟩
Pass ⟨option⟩ to the LLVM assembler (typically clang).

-opta ⟨option⟩
Pass ⟨option⟩ to the assembler.

-optl ⟨option⟩
Pass ⟨option⟩ to the linker.

-optlm ⟨option⟩
Pass ⟨option⟩ to the linker when merging object files. In the case of a standard `ld`-style linker this should generally include the `-r` flag.

-optwindres ⟨option⟩
Pass ⟨option⟩ to `windres` when embedding manifests on Windows. See `-fno-embed-manifest` in *Options affecting linking* (page 245).

-opti ⟨option⟩
Pass ⟨option⟩ to the interpreter sub-process (see *Running the interpreter in a separate process* (page 60)). A common use for this is to pass RTS options e.g., `-opti+RTS -opti-A64m`, or to enable verbosity with `-opti-v` to see what messages are being exchanged by GHC and the interpreter.

So, for example, to force an `-Ewurble` option to the assembler, you would tell the driver `-opta-Ewurble` (the dash before the E is required).

GHC is itself a Haskell program, so if you need to pass options directly to GHC's runtime system you can enclose them in `+RTS ... -RTS` (see *Runtime system (RTS) options* (page 179)).

5.11.3 Options affecting the C pre-processor

CPP

Since 6.8.1

The *CPP* (page 240) language extension enables the C pre-processor. This can be turned into a command-line flag by prefixing it with `-X`; For example:

```
$ ghc -XCPP foo.hs
```

The *CPP* (page 240) language extension can also be enabled using the *LANGUAGE* (page 605) pragma; For example:

```
{-# LANGUAGE CPP #-}
```

-cpp
The C pre-processor **cpp** is run over your Haskell code if the *-cpp* (page 240) option or *CPP* (page 240) extension are given. Unless you are building a large system with significant doses of conditional compilation, you really shouldn't need it.

-D⟨symbol⟩[=⟨value⟩]
Define macro ⟨symbol⟩ in the usual way. When no value is given, the value is taken to be 1. For instance, `-DUSE_MYLIB` is equivalent to `-DUSE_MYLIB=1`.

Note: *-D⟨symbol⟩[=⟨value⟩]* (page 240) does *not* affect `-D` macros passed to the C compiler when compiling an unregisterised build! In this case use the `-optc-Dfoo` hack… (see *Forcing options to a particular phase* (page 239)).

-U⟨symbol⟩
Undefine macro ⟨symbol⟩ in the usual way.

-I⟨dir⟩
Specify a directory in which to look for `#include` files, in the usual C way.

The GHC driver pre-defines several macros when processing Haskell source code (`.hs` or `.lhs` files).

5.11.3.1 Standard CPP macros

The symbols defined by GHC are listed below. To check which symbols are defined by your local GHC installation, the following trick is useful:

```
$ ghc -E -optP-dM -cpp foo.hs
$ cat foo.hspp
```

(you need a file `foo.hs`, but it isn't actually used).

`__GLASGOW_HASKELL__` For version `x.y.z` of GHC, the value of `__GLASGOW_HASKELL__` is the integer ⟨xyy⟩ (if ⟨y⟩ is a single digit, then a leading zero is added, so for example in version 6.2 of GHC, `__GLASGOW_HASKELL__==602`). More information in *GHC version numbering policy* (page 4).

With any luck, `__GLASGOW_HASKELL__` will be undefined in all other implementations that support C-style pre-processing.

Note: The comparable symbols for other systems are: `__HUGS__` for Hugs, `__NHC__` for nhc98, and `__HBC__` for hbc).

NB. This macro is set when pre-processing both Haskell source and C source, including the C source generated from a Haskell module (i.e. `.hs`, `.lhs`, `.c` and `.hc` files).

`__GLASGOW_HASKELL_FULL_VERSION__` This macro exposes the full version string. For instance: `__GLASGOW_HASKELL_FULL_VERSION__==8.11.0.20200319`. Its value comes from the `ProjectVersion` Autotools variable.

Added in GHC 9.0.1

`__GLASGOW_HASKELL_PATCHLEVEL1__`; `__GLASGOW_HASKELL_PATCHLEVEL2__` These macros are available starting with GHC 7.10.1.

For three-part GHC version numbers `x.y.z`, the value of `__GLASGOW_HASKELL_PATCHLEVEL1__` is the integer ⟨z⟩.

For four-part GHC version numbers `x.y.z.z'`, the value of `__GLASGOW_HASKELL_PATCHLEVEL1__` is the integer ⟨z⟩ while the value of `__GLASGOW_HASKELL_PATCHLEVEL2__` is set to the integer ⟨z'⟩.

These macros are provided for allowing finer granularity than is provided by `__GLASGOW_HASKELL__`. Usually, this should not be necessary as it's expected for most APIs to remain stable between patchlevel releases, but occasionally internal API changes are necessary to fix bugs. Also conditional compilation on the patchlevel can be useful for working around bugs in older releases.

Tip: These macros are set when pre-processing both Haskell source and C source, including the C source generated from a Haskell module (i.e. `.hs`, `.lhs`, `.c` and `.hc` files).

`MIN_VERSION_GLASGOW_HASKELL(x,y,z,z')` This macro is available starting with GHC 7.10.1.

This macro is provided for convenience to write CPP conditionals testing whether the GHC version used is version `x.y.z.z'` or later.

If compatibility with Haskell compilers (including GHC prior to version 7.10.1) which do not define MIN_VERSION_GLASGOW_HASKELL is required, the presence of the MIN_VERSION_GLASGOW_HASKELL macro needs to be ensured before it is called, e.g.:

```
#if defined(MIN_VERSION_GLASGOW_HASKELL)
#if MIN_VERSION_GLASGOW_HASKELL(7,10,2,0)
/* code that applies only to GHC 7.10.2 or later */
#endif
#endif
```

Tip: This macro is set when pre-processing both Haskell source and C source, including the C source generated from a Haskell module (i.e. .hs, .lhs, .c and .hc files).

__GLASGOW_HASKELL_TH__ This is set to 1 when the compiler supports Template Haskell, and to 0 when not. The latter is the case for a stage-1 compiler during bootstrapping, or on architectures where the interpreter is not available.

__GLASGOW_HASKELL_LLVM__ Only defined when *:ghc-flag:*-fllvm` is specified. When GHC is using version x.y.z of LLVM, the value of __GLASGOW_HASKELL_LLVM__ is the integer ⟨xyy⟩ (if ⟨y⟩ is a single digit, then a leading zero is added, so for example when using version 3.7 of LLVM, __GLASGOW_HASKELL_LLVM__==307).

__PARALLEL_HASKELL__ Only defined when -parallel is in use! This symbol is defined when pre-processing Haskell (input) and pre-processing C (GHC output).

os_HOST_OS=1 This define allows conditional compilation based on the Operating System, where⟨os⟩ is the name of the current Operating System (eg. linux, mingw32 for Windows, solaris, etc.).

arch_HOST_ARCH=1 This define allows conditional compilation based on the host architecture, where⟨arch⟩ is the name of the current architecture (eg. i386, x86_64, powerpc, sparc, etc.).

VERSION_pkgname This macro is available starting GHC 8.0. It is defined for every exposed package. This macro expands to a string recording the version of pkgname that is exposed for module import. It is identical in behavior to the VERSION_pkgname macros that Cabal defines.

MIN_VERSION_pkgname(x,y,z) This macro is available starting GHC 8.0. It is defined for every exposed package. This macro is provided for convenience to write CPP conditionals testing if a package version is x.y.z or later. It is identical in behavior to the MIN_VERSION_pkgname macros that Cabal defines.

5.11.3.2 CPP and string gaps

A small word of warning: *-cpp* (page 240) is not friendly to "string gaps". In other words, strings such as the following:

```
strmod = "\
\ p \
\ "
```

don't work with *-cpp* (page 240); **/usr/bin/cpp** elides the backslash-newline pairs.

However, it appears that if you add a space at the end of the line, then cpp (at least GNU cpp and possibly other cpps) leaves the backslash-space pairs alone and the string gap works as expected.

5.11.4 Options affecting a Haskell pre-processor

-F
A custom pre-processor is run over your Haskell source file only if the `-F` option is given.

Running a custom pre-processor at compile-time is in some settings appropriate and useful. The `-F` option lets you run a pre-processor as part of the overall GHC compilation pipeline, which has the advantage over running a Haskell pre-processor separately in that it works in interpreted mode and you can continue to take reap the benefits of GHC's recompilation checker.

The pre-processor is run just before the Haskell compiler proper processes the Haskell input, but after the literate markup has been stripped away and (possibly) the C pre-processor has washed the Haskell input.

Use *-pgmF ⟨cmd⟩* (page 238) to select the program to use as the preprocessor. When invoked, the ⟨cmd⟩ pre-processor is given at least three arguments on its command-line: the first argument is the name of the original source file, the second is the name of the file holding the input, and the third is the name of the file where ⟨cmd⟩ should write its output to.

Additional arguments to the pre-processor can be passed in using the *-optF ⟨option⟩* (page 239) option. These are fed to ⟨cmd⟩ on the command line after the three standard input and output arguments.

An example of a pre-processor is to convert your source files to the input encoding that GHC expects, i.e. create a script `convert.sh` containing the lines:

```
#!/bin/sh
( echo "{-# LINE 1 \"$1\" #-}" ; iconv -f l1 -t utf-8 $2 ) > $3
```

and pass `-F -pgmF convert.sh` to GHC. The `-f l1` option tells iconv to convert your Latin-1 file, supplied in argument $2, while the "-t utf-8" options tell iconv to return a UTF-8 encoded file. The result is redirected into argument $3. The `echo "{-# LINE 1 \"$1\" #-}"` just makes sure that your error positions are reported as in the original source file.

5.11.5 Options affecting code generation

-fasm
Use GHC's *native code generator* (page 236) rather than compiling via LLVM. `-fasm` is the default.

-fllvm
Compile via *LLVM* (page 236) instead of using the native code generator. This will generally take slightly longer than the native code generator to compile. Produced code is generally the same speed or faster than the other two code generators. Compiling via LLVM requires LLVM's **opt** and **llc** executables to be in `PATH`.

Note: Note that this GHC release expects an LLVM version between 13 and 16.

-fno-code

Omit code generation (and all later phases) altogether. This is useful if you're only interested in type checking code.

If a module contains a Template Haskell splice then in `--make` mode, code generation will be automatically turned on for all dependencies. By default, object files are generated, but if ghc-flag:*-fprefer-byte-code* is enabled, byte-code will be generated instead.

-fwrite-interface

Always write interface files. GHC will normally write interface files automatically, but this flag is useful with *-fno-code* (page 243), which normally suppresses generation of interface files. This is useful if you want to type check over multiple runs of GHC without compiling dependencies.

-fwrite-if-simplified-core

The interface file will contain all the bindings for a module. From this interface file we can restart code generation to produce byte-code.

The definition of bindings which are included in this depend on the optimisation level. Any definitions which are already included in an interface file (via an unfolding for an exported identifier) are reused.

-fobject-code

Generate object code. This is the default outside of GHCi, and can be used with GHCi to cause object code to be generated in preference to byte-code. Therefore this flag disables *-fbyte-code-and-object-code* (page 244).

-fbyte-code

Generate byte-code instead of object-code. This is the default in GHCi. Byte-code can currently only be used in the interactive interpreter, not saved to disk. This option is only useful for reversing the effect of *-fobject-code* (page 244).

-fbyte-code-and-object-code

Generate object code and byte-code. This is useful with the flags *-fprefer-byte-code* (page 245) and *-fwrite-if-simplified-core* (page 244).

This flag implies *-fwrite-if-simplified-core* (page 244).

-fbyte-code (page 244) and *-fobject-code* (page 244) disable this flag as they specify that GHC should *only* write object code or byte-code respectively.

-fPIC

Generate position-independent code (code that can be put into shared libraries). This currently works on Linux x86 and x86-64. On Windows, position-independent code is never used so the flag is a no-op on that platform.

-fexternal-dynamic-refs

When generating code, assume that entities imported from a different module might be dynamically linked. This flag is enabled automatically by *-dynamic* (page 246).

-fPIE

Generate code in such a way to be linkable into a position-independent executable This currently works on Linux x86 and x86-64. On Windows, position-independent code is never used so the flag is a no-op on that platform. To link the final executable use *-pie* (page 251).

-dynamic

Build code for dynamic linking. This can reduce code size tremendously, but may slowdown cross-module calls of non-inlined functions. There can be some complications combining *-shared* (page 69) with this flag relating to linking in the RTS under Linux. See

#10352.

Note that using this option when linking causes GHC to link against shared libraries.

-dynamic-too

Generates both dynamic and static object files in a single run of GHC. This option is functionally equivalent to running GHC twice, the second time adding `-dynamic -osuf dyn_o -hisuf dyn_hi`.

Although it is equivalent to running GHC twice, using `-dynamic-too` is more efficient, because the earlier phases of the compiler up to code generation are performed just once.

When using `-dynamic-too`, the options `-dyno`, `-dynosuf`, and `-dynhisuf` are the counterparts of `-o`, `-osuf`, and `-hisuf` respectively, but applying to the dynamic compilation.

`-dynamic-too` is ignored if *-dynamic* (page 246) is also specified.

-fexpose-internal-symbols

Request that GHC emits verbose symbol tables which include local symbols for module-internal functions. These can be useful for tools like perf but increase object file sizes. This is implied by *-g2* (page 681) and above.

-fno-expose-internal-symbols (page 245) suppresses all non-global symbol table entries, resulting in smaller object file sizes at the expense of debuggability.

-fprefer-byte-code

If a home package module has byte-code available then use that instead of an object file (if that's available) to evaluate and run TH splices.

This is useful with flags such as *-fbyte-code-and-object-code* (page 244), which tells the compiler to generate byte-code, and *-fwrite-if-simplified-core* (page 244) which allows byte-code to be generated from an interface file.

This flag also interacts with *-fno-code* (page 243), if this flag is enabled then any modules which are required to be compiled for Template Haskell evaluation will generate byte-code rather than object code.

5.11.6 Options affecting linking

GHC has to link your code with various libraries, possibly including: user-supplied, GHC-supplied, and system-supplied (`-lm` math library, for example).

-l ⟨lib⟩

Link in the ⟨lib⟩ library. On Unix systems, this will be in a file called lib*lib*.a or lib*lib*.so which resides somewhere on the library directories path.

Because of the sad state of most UNIX linkers, the order of such options does matter. If library ⟨foo⟩ requires library ⟨bar⟩, then in general `-l ⟨foo⟩` should come *before* `-l ⟨bar⟩` on the command line.

There's one other gotcha to bear in mind when using external libraries: if the library contains a `main()` function, then this will be a link conflict with GHC's own `main()` function (eg. `libf2c` and `libl` have their own `main()`s).

You can use an external main function if you initialize the RTS manually and pass `-no-hs-main`. See also *Using your own main()* (page 569).

-c

Omits the link step. This option can be used with *--make* (page 68) to avoid the automatic linking that takes place if the program contains a `Main` module.

-package ⟨name⟩
If you are using a Haskell "package" (see *Packages* (page 219)), don't forget to add the relevant `-package` option when linking the program too: it will cause the appropriate libraries to be linked in with the program. Forgetting the `-package` option will likely result in several pages of link errors.

-framework ⟨name⟩
On Darwin/OS X/iOS only, link in the framework ⟨name⟩. This option corresponds to the `-framework` option for Apple's Linker. Please note that frameworks and packages are two different things - frameworks don't contain any Haskell code. Rather, they are Apple's way of packaging shared libraries. To link to Apple's "Carbon" API, for example, you'd use `-framework Carbon`.

-staticlib

Implies *-flink-rts* (page 247)

Link all passed files into a static library suitable for linking. To control the name, use the *-o ⟨file⟩* (page 202) option as usual. The default name is `liba.a`.

-L ⟨dir⟩
Where to find user-supplied libraries… Prepend the directory ⟨dir⟩ to the library directories path.

-fuse-rpaths
This flag is enabled by default and will set the rpath of the linked object to the library directories of dependent packages.

When building binaries to distribute it can be useful to pass your own linker options to control the rpath and disable the automatic injection of rpath entries by disabling this flag.

-framework-path ⟨dir⟩
On Darwin/OS X/iOS only, prepend the directory ⟨dir⟩ to the framework directories path. This option corresponds to the `-F` option for Apple's Linker (`-F` already means something else for GHC).

-fsplit-sections
-split-sections
Place each generated function or data item into its own section in the output file if the target supports arbitrary sections. The name of the function or the name of the data item determines the section's name in the output file.

When linking, the linker can automatically remove all unreferenced sections and thus produce smaller executables.

-static
Tell the linker to avoid shared Haskell libraries, if possible. This is the default.

-dynamic
This flag tells GHC to link against shared Haskell libraries. This flag only affects the selection of dependent libraries, not the form of the current target (see *-shared* (page 69)). See *Using shared libraries* (page 251) on how to create them.

Note that this option also has an effect on code generation (see above).

-shared
Instead of creating an executable, GHC produces a shared object with this linker flag. Depending on the operating system target, this might be an ELF DSO, a Windows DLL, or a Mac OS dylib. GHC hides the operating system details beneath this uniform flag.

The flags *-dynamic* (page 246) and *-static* (page 246) control whether the resulting shared object links statically or dynamically to Haskell package libraries given as *-package ⟨pkg⟩* (page 221) option. Non-Haskell libraries are linked as gcc would regularly link it on your system, e.g. on most ELF system the linker uses the dynamic libraries when found.

Object files linked into shared objects must be compiled with *-fPIC* (page 244), see *Options affecting code generation* (page 243)

When creating shared objects for Haskell packages, the shared object must be named properly, so that GHC recognizes the shared object when linking against this package. See *shared object name mangling* (page 231) for details.

-dynload
This flag selects one of a number of modes for finding shared libraries at runtime. See *Finding shared libraries at runtime* (page 253) for a description of each mode.

-flink-rts
When linking shared libraries (*-shared* (page 69)) GHC does not automatically link the RTS. This is to allow choosing the RTS flavour (*-threaded* (page 248), *-eventlog* (page 248), etc) when linking an executable. However when the shared library is the intended product it is useful to be able to reverse this default. See *Shared libraries that export a C API* (page 253) for an usage example.

When linking a static library (*-staticlib* (page 246)) GHC links the RTS automatically, you can reverse this behaviour by reversing this flag: `-fno-link-rts`.

-main-is ⟨thing⟩
The normal rule in Haskell is that your program must supply a `main` function in module `Main`. When testing, it is often convenient to change which function is the "main" one, and the `-main-is` flag allows you to do so. The ⟨thing⟩ can be one of:

- A lower-case identifier `foo`. GHC assumes that the main function is `Main.foo`.
- A module name `A`. GHC assumes that the main function is `A.main`.
- A qualified name `A.foo`. GHC assumes that the main function is `A.foo`.

Strictly speaking, `-main-is` is not a link-phase flag at all; it has no effect on the link step. The flag must be specified when compiling the module containing the specified main function (e.g. module `A` in the latter two items above). It has no effect for other modules, and hence can safely be given to `ghc --make`. However, if all the modules are otherwise up to date, you may need to force recompilation both of the module where the new "main" is, and of the module where the "main" function used to be; `ghc` is not clever enough to figure out that they both need recompiling. You can force recompilation by removing the object file, or by using the *-fforce-recomp* (page 206) flag.

-no-hs-main
In the event you want to include ghc-compiled code as part of another (non-Haskell) program, the RTS will not be supplying its definition of `main()` at link-time, you will have to. To signal that to the compiler when linking, use `-no-hs-main`. See also *Using your own main()* (page 569).

Notice that since the command-line passed to the linker is rather involved, you probably want to use `ghc` to do the final link of your `mixed-language' application. This is not a requirement though, just try linking once with *-v* (page 76) on to see what options the driver passes through to the linker.

The `-no-hs-main` flag can also be used to persuade the compiler to do the link step in *--make* (page 68) mode when there is no Haskell `Main` module present (normally the compiler will not attempt linking when there is no `Main`).

The flags *-rtsopts[=⟨none|some|all|ignore|ignoreAll⟩]* (page 248) and *-with-rtsopts=⟨opts⟩* (page 249) have no effect when used with *-no-hs-main* (page 247), because they are implemented by changing the definition of `main` that GHC generates. See *Using your own main()* (page 569) for how to get the effect of *-rtsopts[=⟨none|some|all|ignore|ignoreAll⟩]* (page 248) and *-with-rtsopts=⟨opts⟩* (page 249) when using your own `main`.

-debug
Link the program with a debugging version of the runtime system. The debugging runtime turns on numerous assertions and sanity checks, and provides extra options for producing debugging output at runtime (run the program with `+RTS -?` to see a list).

-threaded
Link the program with the "threaded" version of the runtime system. The threaded runtime system is so-called because it manages multiple OS threads, as opposed to the default runtime system which is purely single-threaded.

Note that you do *not* need `-threaded` in order to use concurrency; the single-threaded runtime supports concurrency between Haskell threads just fine.

The threaded runtime system provides the following benefits:

- It enables the *-N ⟨x⟩* (page 132) RTS option to be used, which allows threads to run in parallel on a multiprocessor or multicore machine. See *Using SMP parallelism* (page 132).
- If a thread makes a foreign call (and the call is not marked `unsafe`), then other Haskell threads in the program will continue to run while the foreign call is in progress. Additionally, `foreign exported` Haskell functions may be called from multiple OS threads simultaneously. See *Multi-threading and the FFI* (page 573).

-single-threaded

Since 9.8

Switch to the single threaded (default) version of the runtime.

-eventlog

Since Unconditionally enabled with 9.4 and later

Link the program with the "eventlog" version of the runtime system. A program linked in this way can generate a runtime trace of events (such as thread start/stop) to a binary file `program.eventlog`, which can then be interpreted later by various tools. See *Tracing* (page 195) for more information.

Note that as of GHC 9.4 and later eventlog support is included in the RTS by default and the *-eventlog* (page 248) is deprecated.

-rtsopts[=⟨none|some|all|ignore|ignoreAll⟩]

Default some

This option affects the processing of RTS control options given either on the command line or via the *GHCRTS* (page 180) environment variable. There are five possibilities:

-rtsopts=none Disable all processing of RTS options. If `+RTS` appears anywhere on the command line, then the program will abort with an error message. If the `GHCRTS` environment variable is set, then the program will emit a warning message, `GHCRTS` will be ignored, and the program will run as normal.

-rtsopts=ignore Disables all processing of RTS options. Unlike none this treats all RTS flags appearing on the command line the same way as regular arguments. (Passing them on to your program as arguments). GHCRTS options will be processed normally.

-rtsopts=ignoreAll Same as ignore but also ignores GHCRTS.

-rtsopts=some [this is the default setting] Enable only the "safe" RTS options: (Currently only -? and --info.) Any other RTS options on the command line or in the GHCRTS environment variable causes the program with to abort with an error message.

-rtsopts=all or just -rtsopts Enable *all* RTS option processing, both on the command line and through the GHCRTS environment variable.

In GHC 6.12.3 and earlier, the default was to process all RTS options. However, since RTS options can be used to write logging data to arbitrary files under the security context of the running program, there is a potential security problem. For this reason, GHC 7.0.1 and later default to `-rtsopts=some`.

Note that `-rtsopts` has no effect when used with *-no-hs-main* (page 247); see *Using your own main()* (page 569) for details.

`-rtsopts` does not affect RTS options passed via `-with-rtsopts`; those are used regardless of `-rtsopts`.

-with-rtsopts=⟨opts⟩

This option allows you to set the default RTS options at link-time. For example, `-with-rtsopts="-H128m"` sets the default heap size to 128MB. This will always be the default heap size for this program, unless the user overrides it. (Depending on the setting of the `-rtsopts` option, the user might not have the ability to change RTS options at run-time, in which case `-with-rtsopts` would be the *only* way to set them.)

Use the runtime flag *--info* (page 199) on the executable program to see the options set with `-with-rtsopts`.

Note that `-with-rtsopts` has no effect when used with `-no-hs-main`; see *Using your own main()* (page 569) for details.

-no-rtsopts-suggestions

This option disables RTS suggestions about linking with *-rtsopts[=⟨none|some|all|ignore|ignoreAll⟩]* (page 248) when they are not available. These suggestions would be unhelpful if the users have installed Haskell programs through their package managers. With this option enabled, these suggestions will not appear. It is recommended for people distributing binaries to build with either `-rtsopts` or `-no-rtsopts-suggestions`.

-fno-gen-manifest

On Windows, GHC normally generates a manifest file when linking a binary. The manifest is placed in the file *prog*`.exe.manifest`` where ⟨prog.exe⟩ is the name of the executable. The manifest file currently serves just one purpose: it disables the "installer detection" in Windows Vista that attempts to elevate privileges for executables with certain names (e.g. names containing "install", "setup" or "patch"). Without the manifest file to turn off installer detection, attempting to run an executable that Windows deems to be an installer will return a permission error code to the invoker. Depending on the invoker, the result might be a dialog box asking the user for elevated permissions, or it might simply be a permission denied error.

Installer detection can be also turned off globally for the system using the security control panel, but GHC by default generates binaries that don't depend on the user having disabled installer detection.

The `-fno-gen-manifest` disables generation of the manifest file. One reason to do this would be if you had a manifest file of your own, for example.

In the future, GHC might use the manifest file for more things, such as supplying the location of dependent DLLs.

-fno-gen-manifest (page 249) also implies *-fno-embed-manifest* (page 250), see below.

-fno-embed-manifest

The manifest file that GHC generates when linking a binary on Windows is also embedded in the executable itself, by default. This means that the binary can be distributed without having to supply the manifest file too. The embedding is done by running **windres**; to see exactly what GHC does to embed the manifest, use the *-v* (page 76) flag. A GHC installation comes with its own copy of `windres` for this reason.

See also *-pgmwindres ⟨cmd⟩* (page 238) (*Replacing the program for one or more phases* (page 238)) and *-optwindres ⟨option⟩* (page 239) (*Forcing options to a particular phase* (page 239)).

-fno-shared-implib

DLLs on Windows are typically linked to by linking to a corresponding `.lib` or `.dll.a` — the so-called import library. GHC will typically generate such a file for every DLL you create by compiling in *-shared* (page 69) mode. However, sometimes you don't want to pay the disk-space cost of creating this import library, which can be substantial — it might require as much space as the code itself, as Haskell DLLs tend to export lots of symbols.

As long as you are happy to only be able to link to the DLL using `GetProcAddress` and friends, you can supply the *-fno-shared-implib* (page 250) flag to disable the creation of the import library entirely.

-dylib-install-name ⟨path⟩

On Darwin/OS X, dynamic libraries are stamped at build time with an "install name", which is the ultimate install path of the library file. Any libraries or executables that subsequently link against it will pick up that path as their runtime search location for it. By default, ghc sets the install name to the location where the library is built. This option allows you to override it with the specified file path. (It passes `-install_name` to Apple's linker.) Ignored on other platforms.

-rdynamic

This instructs the linker to add all symbols, not only used ones, to the dynamic symbol table. Currently Linux and Windows/MinGW32 only. This is equivalent to using `-optl -rdynamic` on Linux, and `-optl -export-all-symbols` on Windows.

-fwhole-archive-hs-libs

When linking a binary executable, this inserts the flag `-Wl,--whole-archive` before any `-l` flags for Haskell libraries, and `-Wl,--no-whole-archive` afterwards (on OS X, the flag is `-Wl,-all_load`, there is no equivalent for `-Wl,--no-whole-archive`). This flag also disables the use of `-Wl,--gc-sections` (`-Wl,-dead_strip` on OS X).

This is for specialist applications that may require symbols defined in these Haskell libraries at runtime even though they aren't referenced by any other code linked into the executable. If you're using `-fwhole-archive-hs-libs`, you probably also want `-rdynamic`.

-pie

Since 8.2.2

This instructs the linker to produce a position-independent executable. This flag is only valid while producing executables and all object code being linked must have been produced with *-fPIE* (page 244).

Position independent executables are required by some platforms as they enable address-space layout randomization (ASLR), a common security measure. They can also be useful as they can be dynamically loaded and used as shared libraries by other executables.

Position independent executables should be dynamically-linked (e.g. built with *-dynamic* (page 246) and only loaded into other dynamically-linked executables to ensure that only one `libHSrts` is present if loaded into the address space of another Haskell process.

Also, you may need to use the *-rdynamic* (page 250) flag to ensure that that symbols are not dropped from your PIE objects.

-no-pie

If required, the C compiler will still produce a PIE. Otherwise, this is the default. Refer to -pie for more information about PIEs.

-fkeep-cafs

Since 8.8.1

Disables the RTS's normal behaviour of garbage-collecting CAFs (Constant Applicative Forms, in other words top-level expressions). This option is useful for specialised applications that do runtime dynamic linking, where code dynamically linked in the future might require the value of a CAF that would otherwise be garbage-collected.

-fcompact-unwind

Default on

Since 9.4.1

This instructs the linker to produce an executable that supports Apple's compact unwinding sections. These are used by C++ and Objective-C code to unwind the stack when an exception occurs.

In theory, the older *__eh_frame* section should also be usable for this purpose, but this does not always work.

5.12 Using shared libraries

On some platforms GHC supports building Haskell code into shared libraries. Shared libraries are also sometimes known as dynamic libraries, in particular on Windows they are referred to as dynamic link libraries (DLLs).

Shared libraries allow a single instance of some pre-compiled code to be shared between several programs. In contrast, with static linking the code is copied into each program. Using shared libraries can thus save disk space. They also allow a single copy of code to be shared in memory between several programs that use it. Shared libraries are often used as a way of structuring large projects, especially where different parts are written in different programming languages. Shared libraries are also commonly used as a plugin mechanism by various applications. This is particularly common on Windows using COM.

In GHC version 6.12 building shared libraries is supported for Linux (on x86 and x86-64 architectures). GHC version 7.0 adds support on Windows (see *Building and using Win32 DLLs* (page 706)), FreeBSD and OpenBSD (x86 and x86-64), Solaris (x86) and Mac OS X (x86 and PowerPC).

Building and using shared libraries is slightly more complicated than building and using static libraries. When using Cabal much of the detail is hidden, just use `--enable-shared` when configuring a package to build it into a shared library, or to link it against other packages built as shared libraries. The additional complexity when building code is to distinguish whether the code will be used in a shared library or will use shared library versions of other packages it depends on. There is additional complexity when installing and distributing shared libraries or programs that use shared libraries, to ensure that all shared libraries that are required at runtime are present in suitable locations.

5.12.1 Building programs that use shared libraries

To build a simple program and have it use shared libraries for the runtime system and the base libraries use the *-dynamic* (page 246) flag:

```
ghc --make -dynamic Main.hs
```

This has two effects. The first is to compile the code in such a way that it can be linked against shared library versions of Haskell packages (such as base). The second is when linking, to link against the shared versions of the packages' libraries rather than the static versions. Obviously this requires that the packages were built with shared libraries. On supported platforms GHC comes with shared libraries for all the core packages, but if you install extra packages (e.g. with Cabal) then they would also have to be built with shared libraries (`--enable-shared` for Cabal).

5.12.2 Shared libraries for Haskell packages

You can build Haskell code into a shared library and make a package to be used by other Haskell programs. The easiest way is using Cabal, simply configure the Cabal package with the `--enable-shared` flag.

If you want to do the steps manually or are writing your own build system then there are certain conventions that must be followed. Building a shared library that exports Haskell code, to be used by other Haskell code is a bit more complicated than it is for one that exports a C API and will be used by C code. If you get it wrong you will usually end up with linker errors.

In particular Haskell shared libraries *must* be made into packages. You cannot freely assign which modules go in which shared libraries. The Haskell shared libraries must match the package boundaries. The reason for this is that GHC handles references to symbols *within* the same shared library (or main executable binary) differently from references to symbols *between* different shared libraries. GHC needs to know for each imported module if that module lives locally in the same shared lib or in a separate shared lib. The way it does this is by using packages. When using *-dynamic* (page 246), a module from a separate package is assumed to come from a separate shared lib, while modules from the same package (or the default "main" package) are assumed to be within the same shared lib (or main executable binary).

Most of the conventions GHC expects when using packages are described in *Building a package from Haskell source* (page 231). In addition note that GHC expects the `.hi` files to use the extension `.dyn_hi`. The other requirements are the same as for C libraries and are described below, in particular the use of the flags *-dynamic* (page 246), *-fPIC* (page 244) and *-shared* (page 69).

5.12.3 Shared libraries that export a C API

Building Haskell code into a shared library is a good way to include Haskell code in a larger mixed-language project. While with static linking it is recommended to use GHC to perform the final link step, with shared libraries a Haskell library can be treated just like any other shared library. The linking can be done using the normal system C compiler or linker.

It is possible to load shared libraries generated by GHC in other programs not written in Haskell, so they are suitable for using as plugins. Of course to construct a plugin you will have to use the FFI to export C functions and follow the rules about initialising the RTS. See *Making a Haskell library that can be called from foreign code* (page 571). In particular you will probably want to export a C function from your shared library to initialise the plugin before any Haskell functions are called.

To build Haskell modules that export a C API into a shared library use the *-dynamic* (page 246), *-fPIC* (page 244) and *-shared* (page 69) flags:

```
ghc --make -dynamic -shared -fPIC -flink-rts Foo.hs -o libfoo.so
```

As before, the *-dynamic* (page 246) flag specifies that this library links against the shared library versions of the `base` package. *-flink-rts* (page 247) additionally links against the shared library version of the `rts` package (linking against the `rts` package is not enabled by default when building shared libraries). You may also omit `-flink-rts` and link the RTS library into your final executable.

The *-fPIC* (page 244) flag is required for all code that will end up in a shared library. The *-shared* (page 69) flag specifies to make a shared library rather than a program. To make this clearer we can break this down into separate compilation and link steps:

```
ghc -dynamic -fPIC -c Foo.hs
ghc -dynamic -shared -flink-rts Foo.o -o libfoo.so
```

In principle you can use *-shared* (page 69) without *-dynamic* (page 246) in the link step. That means to statically link the runtime system and all of the base libraries into your new shared library. This would make a very big, but standalone shared library. On most platforms however that would require all the static libraries to have been built with *-fPIC* (page 244) so that the code is suitable to include into a shared library and we do not do that at the moment.

> **Warning:** If your shared library exports a Haskell API then you cannot directly link it into another Haskell program and use that Haskell API. You will get linker errors. You must instead make it into a package as described in the section above.

5.12.4 Finding shared libraries at runtime

The primary difficulty with managing shared libraries is arranging things such that programs can find the libraries they need at runtime. The details of how this works varies between platforms, in particular the three major systems: Unix ELF platforms, Windows and Mac OS X.

5.12.4.1 Unix

On Unix there are two mechanisms. Shared libraries can be installed into standard locations that the dynamic linker knows about. For example `/usr/lib` or `/usr/local/lib` on most systems. The other mechanism is to use a “runtime path” or “rpath” embedded into programs and libraries themselves. These paths can either be absolute paths or on at least Linux and Solaris they can be paths relative to the program or library itself. In principle this makes it possible to construct fully relocatable sets of programs and libraries.

GHC has a `-dynload` linking flag to select the method that is used to find shared libraries at runtime. There are currently two modes:

`sysdep` A system-dependent mode. This is also the default mode. On Unix ELF systems this embeds `RPATH`/`RUNPATH` entries into the shared library or executable. In particular it uses absolute paths to where the shared libraries for the rts and each package can be found. This means the program can immediately be run and it will be able to find the libraries it needs. However it may not be suitable for deployment if the libraries are installed in a different location on another machine.

`deploy` This does not embed any runtime paths. It relies on the shared libraries being available in a standard location or in a directory given by the `LD_LIBRARY_PATH` environment variable.

To use relative paths for dependent libraries on Linux and Solaris you can pass a suitable `-rpath` flag to the linker:

```
ghc -dynamic Main.hs -o main -lfoo -L. -optl-Wl,-rpath,'$ORIGIN'
```

This assumes that the library `libfoo.so` is in the current directory and will be able to be found in the same directory as the executable `main` once the program is deployed. Similarly it would be possible to use a subdirectory relative to the executable e.g. `-optl-Wl,-rpath,'$ORIGIN/lib'`.

This relative path technique can be used with either of the two `-dynload` modes, though it makes most sense with the `deploy` mode. The difference is that with the `deploy` mode, the above example will end up with an ELF `RUNPATH` of just `$ORIGIN` while with the `sysdep` mode the `RUNPATH` will be `$ORIGIN` followed by all the library directories of all the packages that the program depends on (e.g. `base` and `rts` packages etc.) which are typically absolute paths. The unix tool `readelf --dynamic` is handy for inspecting the `RPATH`/`RUNPATH` entries in ELF shared libraries and executables.

On most UNIX platforms it is also possible to build executables that can be `dlopen`’d like shared libraries using the *`-pie`* (page 251) flag during linking.

5.12.4.2 Mac OS X

The standard assumption on Darwin/Mac OS X is that dynamic libraries will be stamped at build time with an “install name”, which is the full ultimate install path of the library file. Any libraries or executables that subsequently link against it (even if it hasn’t been installed yet) will pick up that path as their runtime search location for it. When compiling with ghc directly, the install name is set by default to the location where it is built. You can override this with the *`-dylib-install-name ⟨path⟩`* (page 250) option (which passes `-install_name` to the Apple linker). Cabal does this for you. It automatically sets the install name for dynamic libraries to the absolute path of the ultimate install location.

5.13 Debugging the compiler

HACKER TERRITORY. HACKER TERRITORY. (You were warned.)

Dump flags

5.13.1 Dumping out compiler intermediate structures

`-ddump-to-file`

Causes the output from each of flags starting with "-ddump", to be dumped to a file or files. If you want to have all the output from one single flag saved to one file, use *`-ddump-file-prefix=⟨str⟩`* (page 255) (see descriptions below). Otherwise, the output will go to several files, including one for non-module specific and several for module specific. The suffix of a dump file depends on the flag turned on, for instance, output from *`-ddump-simpl`* (page 259) will end up in `prefix.dump-simpl`.

`-ddump-file-prefix`=⟨str⟩

Set the prefix of the filenames used for debugging output. For example, `-ddump-file-prefix=Foo` will cause the output from *`-ddump-simpl`* (page 259) to be dumped to `Foo.dump-simpl`.

`-fdump-with-ways`

Default enabled

When compiling Main.hs with profiling and without this will now produce `Main.p.dump-simpl` and `Main.dump-simpl` instead of overwriting the output of one way with the output of another.

-ddump-json
This flag was previously used to generated JSON formatted GHC diagnostics, but has been deprecated. Instead, use *-fdiagnostics-as-json* (page 80).

-dshow-passes
Print out each pass name, its runtime and heap allocations as it happens. Note that this may come at a slight performance cost as the compiler will be a bit more eager in forcing pass results to more accurately account for their costs.

Two types of messages are produced: Those beginning with *** do denote the beginning of a compilation phase whereas those starting with `!!!` mark the end of a pass and are accompanied by allocation and runtime statistics.

-dipe-stats
For each module, show some simple statistics about which info tables have IPE information, and how many info tables with IPE information each closure type has. This is useful, for example, for verifying that `STACK` info tables are being appropriately omitted or included from the info table map.

-dfaststring-stats
Show statistics on the usage of fast strings by the compiler.

-ddump-faststrings
Dump the whole FastString table when finished. Consider using *-ddump-file-prefix=⟨str⟩* (page 255) to dump it into a file.

-dppr-debug
Debugging output is in one of several "styles." Take the printing of types, for example. In the "user" style (the default), the compiler's internal ideas about types are presented in Haskell source-level syntax, insofar as possible. In the "debug" style (which is the default for debugging output), the types are printed in with explicit foralls, and variables have their unique-id attached (so you can check for things that look the same but aren't). This flag makes debugging output appear in the more verbose debug style.

-ddump-timings
Show allocation and runtime statistics for various stages of compilation. Allocations are measured in bytes. Timings are measured in milliseconds.

GHC is a large program consisting of a number of stages. You can tell GHC to dump information from various stages of compilation using the `-ddump-⟨pass⟩` flags listed below. Note that some of these tend to produce a lot of output. You can prevent them from clogging up your standard output by passing *-ddump-to-file* (page 255).

5.13.1.1 Front-end

These flags dump various information from GHC's frontend. This includes the parser and interface file reader.

-ddump-parsed
Dump parser output

-ddump-parsed-ast
Dump parser output as a syntax tree

-dkeep-comments
Include comments in the parser. Useful in combination with *-ddump-parsed-ast* (page 256).

-ddump-if-trace
Make the interface loader be *real* chatty about what it is up to.

5.13.1.2 Type-checking and renaming

These flags dump various information from GHC's typechecker and renamer.

-ddump-tc-trace
Make the type checker be *real* chatty about what it is up to.

-ddump-rn-trace
Make the renamer be *real* chatty about what it is up to.

-ddump-ec-trace
Make the pattern match exhaustiveness checker be *real* chatty about what it is up to.

-ddump-cs-trace
Make the constraint solver be *real* chatty about what it is up to.

-ddump-rn-stats
Print out summary of what kind of information the renamer had to bring in.

-ddump-rn
Dump renamer output

-ddump-rn-ast
Dump renamer output as a syntax tree

-ddump-tc
Dump typechecker output. Note that this hides a great deal of detail by default; you might consider using this with *-fprint-typechecker-elaboration* (page 79).

-ddump-tc-ast
Dump typechecker output as a syntax tree

-ddump-hie
Dump the hie file syntax tree if we are generating extended interface files

-ddump-splices
Dump Template Haskell expressions that we splice in, and what Haskell code the expression evaluates to.

-dth-dec-file
Dump expansions of all top-level Template Haskell splices into `module.th.hs` for each file `module.hs`.

-ddump-types
Dump a type signature for each value defined at the top level of the module. The list is sorted alphabetically. Using *-dppr-debug* (page 256) dumps a type signature for all the imported and system-defined things as well; useful for debugging the compiler.

-ddump-deriv
Dump derived instances

5.13.1.3 Core representation and simplification

These flags dump various phases of GHC's Core-to-Core pipeline. This begins with the desugarer and includes the simplifier, worker-wrapper transformation, the rule engine, the specialiser, the strictness/occurrence analyser, and a common subexpression elimination pass.

-ddump-call-arity
: Dump output of the call arity analysis pass (*-fcall-arity* (page 114)).

-ddump-core-stats
: Print a one-line summary of the size of the Core program at the end of the optimisation pipeline.

-ddump-ds
-ddump-ds-preopt
: Dump desugarer output. *-ddump-ds* (page 258) dumps the output after the very simple optimiser has run (which discards a lot of clutter and hence is a sensible default. *-ddump-ds-preopt* (page 258) shows the output after desugaring but before the very simple optimiser.

-ddump-exitify
: Dump output of the exitification pass (*-fexitification* (page 114)), which tries to pull out code out of recursive functions.

-ddump-simpl-iterations
: Show the output of each *iteration* of the simplifier (each run of the simplifier has a maximum number of iterations, normally 4).

-ddump-simpl-stats
: Dump statistics about how many of each kind of transformation took place. If you add *-dppr-debug* (page 256) you get more detailed information.

-ddump-simpl-trace
: Dump trace messages from various functions of the simplifier. Produces quite a lot of output.

-dverbose-core2core
: Show the output of the intermediate Core-to-Core pass. (*lots* of output!) So: when we're really desperate:

```
% ghc -noC -O -ddump-simpl -dverbose-core2core -dcore-lint Foo.hs
```

-ddump-spec
: Dump output of typeclass specialisation pass

-ddump-spec-constr

Since 9.8.1

: Dump output of the SpecConstr specialisation pass

-ddump-rules
: Dumps all rewrite rules specified in this module; see *Controlling what's going on in rewrite rules* (page 596).

-ddump-rule-firings
: Dumps the names of all rules that fired in this module

-ddump-rule-rewrites
: Dumps detailed information about all rules that fired in this module

-drule-check=⟨str⟩
This flag is useful for debugging why a rule you expect to be firing isn't.

Rules are filtered by the user provided string, a rule is kept if a prefix of its name matches the string. The pass then checks whether any of these rules could apply to the program but which didn't fire for some reason. For example, specifying `-drule-check=SPEC` will check whether there are any applications which might be subject to a rule created by specialisation.

-dinline-check=⟨str⟩
This flag is useful for debugging why a definition is not inlined.

When a string is passed to this flag we report information about all functions whose name shares a prefix with the string.

For example, if you are inspecting the core of your program and you observe that `foo` is not being inlined. You can pass `-dinline-check foo` and you will see a report about why `foo` is not inlined.

-ddump-simpl
Dump simplifier output (Core-to-Core passes)

-ddump-inlinings
Dumps inlinings performed by the simplifier.

-ddump-verbose-inlinings
Dumps all inlinings considered by the simplifier, even those ultimately not performed. This output includes various information that the simplifier uses to determine whether the inlining is beneficial.

-ddump-stranal
Has been renamed to *-ddump-dmdanal* (page 259).

-ddump-dmdanal
Dump demand analysis output.

See *-fstrictness* (page 126) for the syntax and semantics of demand annotations.

-ddump-str-signatures
Has been renamed to *-ddump-dmd-signatures* (page 259).

-ddump-dmd-signatures
Dump top-level demand signatures as produced by demand analysis.

See *-fstrictness* (page 126) for the syntax and semantics of demand annotations.

-ddump-cpranal
Dump Constructed Product Result analysis output

-ddump-cpr-signatures
Dump Constructed Product Result signatures

-ddump-cse
Dump common subexpression elimination (CSE) pass output

-ddump-full-laziness
-ddump-float-out
Dump full laziness pass (also known as float-out) output (see *-ffull-laziness* (page 118))

-ddump-float-in
Dump float-in pass output (see *-ffloat-in* (page 118))

-ddump-liberate-case
Dump liberate case pass output (see *-fliberate-case* (page 119))

-ddump-static-argument-transformation
Dump static argument transformation pass output (see *-fstatic-argument-transformation* (page 125))

-ddump-worker-wrapper
Dump worker/wrapper split output

-ddump-occur-anal
Dump "occurrence analysis" output

-ddump-prep
Dump output of Core preparation pass

-ddump-late-cc
Dump output of LateCC pass after cost centres have been added.

-ddump-view-pattern-commoning
Print the view patterns that are commoned.

5.13.1.4 STG representation

These flags dump various phases of GHC's STG pipeline.

-ddump-stg-from-core
Show the output of CoreToStg pass.

-dverbose-stg2stg
Show the output of the intermediate STG-to-STG pass. (*lots* of output!)

-ddump-stg-unarised
Show the output of the unarise pass.

-ddump-stg-cg
Show the output of the STG after Stg2Stg. This is the result after applying the Stg2Stg optimization passes.

-ddump-stg-tags
Show the output of the tag inference pass.

-ddump-stg-final
Show the output of the last STG pass before we generate Cmm.

-ddump-stg
Alias for *-ddump-stg-from-core* (page 260). Deprecated in favor of more explicit flags: *-ddump-stg-from-core* (page 260), *-ddump-stg-final* (page 260), etc.

5.13.1.5 C-- representation

These flags dump various phases of GHC's C-- pipeline.

-ddump-cmm-verbose-by-proc
Dump output from main C-- pipeline stages. In case of `.cmm` compilation this also dumps the result of file parsing. Not included are passes run by the chosen backend. Currently only the NCG backends runs additional passes (*-ddump-opt-cmm* (page 262)).

Cmm dumps don't include unreachable blocks since we print blocks in reverse post-order.

-ddump-cmm-verbose
If used in conjunction with *-ddump-to-file* (page 255), writes dump output from main C-- pipeline stages to files (each stage per file).

-ddump-cmm-from-stg
Dump the result of STG-to-C-- conversion

-ddump-cmm-raw
Dump the "raw" C--.

-ddump-cmm-cfg
Dump the results of the C-- control flow optimisation pass.

-ddump-cmm-thread-sanitizer
Dump the results of the C-- pass responsible for adding instrumentation added by *-fcmm-thread-sanitizer* (page 266).

-ddump-cmm-cbe
Dump the results of the C-- Common Block Elimination (CBE) pass.

-ddump-cmm-switch
Dump the results of the C-- switch lowering pass.

-ddump-cmm-proc
Dump the results of the C-- proc-point analysis pass.

-ddump-cmm-sp
Dump the results of the C-- stack layout pass.

-ddump-cmm-sink
Dump the results of the C-- sinking pass.

-ddump-cmm-caf
Dump the results of the C-- CAF analysis pass.

-ddump-cmm-procmap
Dump the results of the C-- proc-point map pass.

-ddump-cmm-split
Dump the results of the C-- proc-point splitting pass.

-ddump-cmm-info
Dump the results of the C-- info table augmentation pass.

-ddump-cmm-cps
Dump the results of the CPS pass.

-ddump-cmm
Dump the result of the C-- pipeline processing

-ddump-cfg-weights
Dumps the CFG with weights used by the new block layout code. Each CFG is dumped in dot format graph making it easy to visualize them.

5.13.1.6 LLVM code generator

-ddump-llvm

Implies *-fllvm* (page 243)

LLVM code from the *LLVM code generator* (page 236)

5.13.1.7 C code generator

-ddump-c-backend

Shortdesc Dump C code produced by the C (unregisterised) backend.

5.13.1.8 Native code generator

These flags dump various stages of the *native code generator's* (page 236) pipeline, which starts with C-- and produces native assembler.

-ddump-cmm-opt
Dump the results of C-- to C-- optimising passes performed by the NCG.

-ddump-opt-cmm
Alias for *-ddump-cmm-opt* (page 262)

-ddump-asm-conflicts
Dump (virtual) register conflicts ("interferences") from the graph coloring register allocator (*-fregs-graph* (page 122)).

-ddump-asm-native
Dump the initial assembler output produced from C--.

-ddump-asm-liveness
Dump the result of the register liveness pass.

-ddump-asm-regalloc
Dump the result of the register allocation pass.

-ddump-asm-regalloc-stages
Dump the build/spill stages of the *-fregs-graph* (page 122) register allocator.

-ddump-asm-stats
Dump statistics from the register allocator.

-ddump-asm
Dump the final assembly produced by the native code generator.

-ddump-js
Dump the final JavaScript code produced by the JavaScript code generator.

5.13.1.9 JavaScript code generator

-ddisable-js-minifier
Include human-readable spacing and indentation when generating JavaScript.

-ddisable-js-c-sources
For debugging it can be useful to avoid linking with C sources compiled to JavaScript with Emscripten. This also avoids linking with Emcscripten's RTS. Note that code that calls into this C code or that uses Emscripten's primitives will fail at runtime (e.g. undefined function errors).

5.13.1.10 Miscellaneous backend dumps

These flags dump various bits of information from other backends.

-ddump-bcos
Dump byte-code objects (BCOs) produced for the GHC's byte-code interpreter.

-ddump-debug
Dump generated debug information (DWARF) produced with the *-g* (page 681) flag.

-ddump-rtti
Trace runtime type inference done by various interpreter commands.

-ddump-foreign
Dump foreign export stubs.

-ddump-ticked
Dump the code instrumented by HPC (*Observing Code Coverage* (page 672)).

-ddump-hpc
An alias for *-ddump-ticked* (page 263).

-ddump-mod-map
Dump a mapping of modules to where they come from, and how:

- `(hidden module)`: Module is hidden, and thus will never be available for import.
- `(unusable module)`: Module is unavailable because the package is unusable.
- `(hidden package)`: This module is in someone's exported-modules list, but that package is hidden.
- `(exposed package)`: Module is available for import.
- `(reexport by <PACKAGES>)`: This module is available from a reexport of some set of exposed packages.
- `(hidden reexport by <PACKAGES>)`: This module is available from a reexport of some set of hidden packages.
- `(package flag)`: This module export comes from a package flag.

5.13.2 Formatting dumps

-dppr-user-length
In error messages, expressions are printed to a certain "depth", with subexpressions beyond the depth replaced by ellipses. This flag sets the depth. Its default value is 5.

-dppr-cols=⟨n⟩
Set the width of debugging output. Use this if your code is wrapping too much. For example: `-dppr-cols=200`.

-dppr-case-as-let
Print single alternative case expressions as though they were strict let expressions. This is helpful when your code does a lot of unboxing.

-dhex-word-literals
Print values of type *Word#* and *Word64#* (but not values of type *Int#* and *Int64#*) in hexadecimal instead of decimal. The hexadecimal is zero-padded to make the length of the representation a power of two. For example: *0x0A0A##*, *0x000FFFFF##*, *0xC##*. This flag may be helpful when you are producing a bit pattern that to expect to work correctly on a 32-bit or a 64-bit architecture. Dumping hexadecimal literals after optimizations and constant folding makes it easier to confirm that the generated bit pattern is correct.

-dno-debug-output
Suppress any unsolicited debugging output. When GHC has been built with the `DEBUG` option it occasionally emits debug output of interest to developers. The extra output can confuse the testing framework and cause bogus test failures, so this flag is provided to turn it off.

5.13.3 Suppressing unwanted information

Core dumps contain a large amount of information. Depending on what you are doing, not all of it will be useful. Use these flags to suppress the parts that you are not interested in.

-dsuppress-all
Suppress everything that can be suppressed, except for unique ids as this often makes the printout ambiguous. If you just want to see the overall structure of the code, then start here.

-dsuppress-ticks
Suppress "ticks" in the pretty-printer output.

-dsuppress-uniques
Suppress the printing of uniques. This may make the printout ambiguous (e.g. unclear where an occurrence of 'x' is bound), but it makes the output of two compiler runs have many fewer gratuitous differences, so you can realistically apply `diff`. Once `diff` has shown you where to look, you can try again without *-dsuppress-uniques* (page 264)

-dsuppress-idinfo
Suppress extended information about identifiers where they are bound. This includes strictness information and inliner templates. Using this flag can cut the size of the core dump in half, due to the lack of inliner templates

-dsuppress-unfoldings
Suppress the printing of the stable unfolding of a variable at its binding site.

-dsuppress-module-prefixes
Suppress the printing of module qualification prefixes. This is the `Data.List` in `Data.List.length`.

-dsuppress-timestamps
Suppress the printing of timestamps. This makes it easier to diff dumps.

-dsuppress-type-signatures
Suppress the printing of type signatures.

-dsuppress-type-applications
Suppress the printing of type applications.

-dsuppress-coercions
Suppress the printing of type coercions.

-dsuppress-coercion-types

-dsuppress-var-kinds
Suppress the printing of variable kinds

-dsuppress-stg-free-vars
Suppress the printing of closure free variable lists in STG output

-dsuppress-core-sizes

Since 9.4.1

Suppress the printing of core size stats per binding

-dsuppress-stg-reps

Since 9.6.1

default: enabled

Disabling this will annoate certain stg arguments with their prim rep.

5.13.4 Checking for consistency

-dlint

Implies -dcore-lint, -dstg-lint, -dcmm-lint, -dasm-lint, -fllvm-fill-undef-with-garbage, -fcatch-nonexhaustive-cases, -debug

Since 9.4.1

Turn on various heavy-weight intra-pass sanity-checking measures within GHC and its runtime system. Notably, this does not include *-falignment-sanitisation* (page 266) as it incurs a rather hefty runtime cost.

-dcore-lint
Turn on heavyweight intra-pass sanity-checking within GHC, at Core level. (It checks GHC's sanity, not yours.)

-dlinear-core-lint
Turn on linearity checking in GHC. Currently, some optimizations in GHC might not preserve linearity and there are valid programs that fail Linear Core Lint. In the near future, this option will be removed and folded into normal Core Lint.

-dstg-lint
Ditto for STG level.

-dcmm-lint
Ditto for C-- level.

-dasm-lint
Turn on intra-pass sanity-checking within GHC, at the code generator level.

-fllvm-fill-undef-with-garbage
Instructs the LLVM code generator to fill dead STG registers with garbage instead of `undef` in calls. This makes it easier to catch subtle code generator and runtime system bugs (e.g. see #11487).

-falignment-sanitisation
Compile with alignment checks for all info table dereferences. This can be useful when finding pointer tagging issues.

-fproc-alignment

Since 8.6.1

Align functions to multiples of the given value. Only valid values are powers of two.

`-fproc-alignment=64` can be used to limit alignment impact on performance as each function will start at a cache line. However forcing larger alignments in general reduces performance.

-fcatch-nonexhaustive-cases
GHC generates case expressions without a default alternative in some cases:

- When the demand analysis thinks that the scrutinee does not return (i.e. a bottoming expression)
- When the scrutinee is a GADT and its type rules out some constructors, and others constructors are already handled by the case expression.

With this flag GHC generates a default alternative with `error` in these cases. This is helpful when debugging demand analysis or type checker bugs which can sometimes manifest as segmentation faults.

-forig-thunk-info
When debugging cyclic thunks it can be helpful to know the original info table of a thunk being evaluated. This flag enables code generation logic to facilitate this, producing a `stg_orig_thunk_info` stack frame alongside the usual update frame; such `orig_thunk` frames have no operational effect but capture the original info table of the updated thunk for inspection by debugging tools. See `Note [Original thunk info table frames]` in `GHC.StgToCmm.Bind` for details.

-fcheck-prim-bounds
Typically primops operations like `writeArray#` exhibit unsafe behavior, relying on the user to perform any bounds checking. This flag instructs the code generator to instrument such operations with bound checking logic which aborts the program when an out-of-bounds access is detected.

Note that this is only intended to be used as a debugging measure, not as the primary means of catching out-of-bounds accesses.

-fcmm-thread-sanitizer
This enables generation of *ThreadSanitizer <https://github.com/google/sanitizers/wiki/ThreadSanitizer* instrumentation of memory accesses. Requires use of `-fsanitize=thread` or similar when compiling and linking.

5.13.5 Checking for determinism

-dinitial-unique=⟨s⟩
Start `UniqSupply` allocation from ⟨s⟩.

-dunique-increment=⟨i⟩
Set the increment for the generated `Unique`'s to ⟨i⟩.

This is useful in combination with *-dinitial-unique=⟨s⟩* (page 267) to test if the generated files depend on the order of `Unique`'s.

Some interesting values:

- `-dinitial-unique=0 -dunique-increment=1` - current sequential `UniqSupply`
- `-dinitial-unique=16777215 -dunique-increment=-1` - `UniqSupply` that generates in decreasing order
- `-dinitial-unique=1 -dunique-increment=PRIME` - where PRIME big enough to overflow often - nonsequential order

5.13.6 Other

-dno-typeable-binds
This avoids generating Typeable-related bindings for modules and types. This is useful when debugging because it gives smaller modules and dumps, but the compiler will panic if you try to use Typeable instances of things that you built with this flag.

-dtag-inference-checks
When tag inference tells as a specific value is supposed to be tagged then generate code to check this at runtime. If the check fails the program will be terminated. This helps narrowing down if an issue is due to tag inference if things go wrong. Which would otherwise be quite difficult.

-funoptimized-core-for-interpreter

Since 9.8.1

default: enabled

At the moment, ghci disables optimizations, because not all passes are compatible with the interpreter. This option can be used to override this check, e.g. `ghci -O2 -fno-unoptimized-core-for-interpreter`. It is not recommended for normal use and can cause a compiler panic.

Note that this has an effect on the debugger interface: With optimizations in play, free variables in breakpoints may now be substituted with complex expressions. Those cannot be stored in breakpoints, so any free variable that refers to optimized code will not be inspectable when this flag is enabled.

CHAPTER

SIX

LANGUAGE EXTENSIONS

6.1 Introduction

GHC implements several variants of the Haskell language, along with many extensions. They can all be enabled or disabled by command line flags or *LANGUAGE* (page 605) pragmas.

Some of the extensions serve to give you access to the underlying facilities with which we implement Haskell. Thus, you can get at the Raw Iron, if you are willing to write some non-portable code at a more primitive level. You need not be "stuck" on performance because of the implementation costs of Haskell's "high-level" features—you can always code "under" them. In an extreme case, you can write all your time-critical code in C, and then just glue it together with Haskell!

6.1.1 Controlling editions and extensions

GHC supports multiple language editions: *Haskell98* (page 272), *Haskell2010* (page 272), *GHC2021* (page 271) and *GHC2024* (page 270). Each language edition consists of a collection of language extensions, and there are many other language extensions not currently part of a language edition but that can be enabled explicitly.

Currently, *GHC2021* (page 271) is used by default if no other language edition is explicitly requested, for backwards compatibility purposes. Since later versions of GHC may use a different language edition by default, users are advised to declare a language edition explicitly. Using *GHC2024* (page 270) is recommended for new code.

A language edition can be selected:

- at the package level, e.g. using `default-language: GHC2024` in a `.cabal` file;
- with a command-line flag prefixed by "`-X...`" (e.g. `-XGHC2024`); or
- for an individual module using the *LANGUAGE* (page 605) pragma, e.g. `{-# LANGUAGE GHC2024 #-}`.

Selecting a language edition overrides any previous selection. It is not possible to disable a language edition.

Similarly, language extensions can be controlled (either enabled or disabled):

- at the package level, e.g. using `default-extensions: TemplateHaskell` in a `.cabal` file;
- with command-line flags, switched on by a command-line flag "`-X...`" (e.g. `-XTemplateHaskell`), and switched off by the flag "`-XNo...`"; (e.g. `-XNoTemplateHaskell`);

- for an individual module using the *LANGUAGE* (page 605) pragma, e.g. `{-# LANGUAGE TemplateHaskell #-}` or `{-# LANGUAGE NoTemplateHaskell #-}`.

GHC2024

Since 9.10.1

GHC blesses a number of extensions, beyond Haskell 2010, to be suitable to turned on by default. These extensions are considered to be stable and conservative.

Note that, because GHC2024 includes a number of non-standardized extensions, the stability guarantees it provides are not quite as strong as those provided by, e.g., *Haskell2010* (page 272). While GHC does take pains to avoid changing the semantics of these extensions, changes may still happen (e.g. the simplified subsumption change introduced in GHC 9.0 which caused GHC to reject some programs using *RankNTypes* (page 402)).

The `GHC2024` language edition includes the following extensions:

- *BangPatterns* (page 541)
- *BinaryLiterals* (page 491)
- *ConstrainedClassMethods* (page 471)
- *ConstraintKinds* (page 500)
- *DataKinds* (page 357)
- *DeriveDataTypeable* (page 445)
- *DeriveFoldable* (page 442)
- *DeriveFunctor* (page 439)
- *DeriveGeneric* (page 598)
- *DeriveLift* (page 446)
- *DeriveTraversable* (page 444)
- *DerivingStrategies* (page 455)
- *DisambiguateRecordFields* (page 423)
- `DoAndIfThenElse`
- *EmptyCase* (page 300)
- *EmptyDataDecls* (page 321)
- *EmptyDataDeriving* (page 435)
- *ExistentialQuantification* (page 325)
- *ExplicitForAll* (page 506)
- *ExplicitNamespaces* (page 319)
- *FieldSelectors* (page 426)
- *FlexibleContexts* (page 499)
- *FlexibleInstances* (page 480)
- *ForeignFunctionInterface* (page 561)
- *GADTs* (page 335)
- *GADTSyntax* (page 329)
- *GeneralisedNewtypeDeriving* (page 447)
- *HexFloatLiterals* (page 492)
- *ImplicitPrelude* (page 295)
- *ImportQualifiedPost* (page 321)
- *InstanceSigs* (page 490)
- *KindSignatures* (page 511)
- *LambdaCase* (page 299)
- *MonoLocalBinds* (page 526)
- *MonomorphismRestriction* (page 525)
- *MultiParamTypeClasses* (page 470)
- *NamedFieldPuns* (page 427)
- *NamedWildCards* (page 522)

- *NumericUnderscores* (page 494)
- *PatternGuards* (page 459)
- *PolyKinds* (page 364)
- *PostfixOperators* (page 298)
- *RankNTypes* (page 402)
- RelaxedPolyRec
- *RoleAnnotations* (page 419)
- *ScopedTypeVariables* (page 512)
- *StandaloneDeriving* (page 436)
- *StandaloneKindSignatures* (page 369)
- *StarIsType* (page 378)
- *TraditionalRecordSyntax* (page 421)
- *TupleSections* (page 298)
- *TypeApplications* (page 386)
- *TypeOperators* (page 323)
- *TypeSynonymInstances* (page 480)

GHC2021

Since 9.2.1

See *GHC2024* (page 270) for general comments about GHC20xx language editions.

Also note that due to a minor oversight, enabling this edition behaves slightly differently than enabling each of its constituent extensions. Specifically, while *TypeOperators* (page 323) implies *ExplicitNamespaces* (page 319), *ExplicitNamespaces* (page 319) is not included in *GHC2021* (page 271). Moreover, while *GADTs* (page 335) is not part of *GHC2021* (page 271), the combination of *GADTSyntax* (page 329) and *ExistentialQuantification* (page 325) is enough to define and use GADTs.

The GHC2021 language edition includes the following extensions:

- *BangPatterns* (page 541)
- *BinaryLiterals* (page 491)
- *ConstrainedClassMethods* (page 471)
- *ConstraintKinds* (page 500)
- *DeriveDataTypeable* (page 445)
- *DeriveFoldable* (page 442)
- *DeriveFunctor* (page 439)
- *DeriveGeneric* (page 598)
- *DeriveLift* (page 446)
- *DeriveTraversable* (page 444)
- DoAndIfThenElse
- *EmptyCase* (page 300)
- *EmptyDataDecls* (page 321)
- *EmptyDataDeriving* (page 435)
- *ExistentialQuantification* (page 325)
- *ExplicitForAll* (page 506)
- *FieldSelectors* (page 426)
- *FlexibleContexts* (page 499)
- *FlexibleInstances* (page 480)
- *ForeignFunctionInterface* (page 561)
- *GADTSyntax* (page 329)
- *GeneralisedNewtypeDeriving* (page 447)
- *HexFloatLiterals* (page 492)
- *ImplicitPrelude* (page 295)
- *ImportQualifiedPost* (page 321)

- *InstanceSigs* (page 490)
- *KindSignatures* (page 511)
- *MonomorphismRestriction* (page 525)
- *MultiParamTypeClasses* (page 470)
- *NamedFieldPuns* (page 427)
- *NamedWildCards* (page 522)
- *NumericUnderscores* (page 494)
- *PatternGuards* (page 459)
- *PolyKinds* (page 364)
- *PostfixOperators* (page 298)
- *RankNTypes* (page 402)
- RelaxedPolyRec
- *ScopedTypeVariables* (page 512)
- *StandaloneDeriving* (page 436)
- *StandaloneKindSignatures* (page 369)
- *StarIsType* (page 378)
- *TraditionalRecordSyntax* (page 421)
- *TupleSections* (page 298)
- *TypeApplications* (page 386)
- *TypeOperators* (page 323)
- *TypeSynonymInstances* (page 480)
- *NoExplicitNamespaces* (page 319)

Haskell2010

Compile using the Haskell 2010 language edition, as specified by the Haskell 2010 report. GHC aims to behave mostly as a Haskell 2010 compiler, but there are a few known deviations from the standard (see *Haskell standards vs. Glasgow Haskell: language non-compliance* (page 731)).

The Haskell2010 language edition includes the following language extensions:

- *CUSKs* (page 367)
- *DatatypeContexts* (page 322)
- *DeepSubsumption* (page 405)
- DoAndIfThenElse
- *EmptyDataDecls* (page 321)
- *FieldSelectors* (page 426)
- *ForeignFunctionInterface* (page 561)
- *ImplicitPrelude* (page 295)
- *MonomorphismRestriction* (page 525)
- *PatternGuards* (page 459)
- RelaxedPolyRec
- *StarIsType* (page 378)
- *TraditionalRecordSyntax* (page 421)

Haskell98

Compile using the Haskell 98 language edition, as specified by the Haskell 98 report. GHC aims to behave mostly as a Haskell 98 compiler, but there are a few known deviations from the standard (see *Haskell standards vs. Glasgow Haskell: language non-compliance* (page 731)).

The Haskell98 language edition includes the following language extensions:

- *CUSKs* (page 367)
- *DatatypeContexts* (page 322)
- *DeepSubsumption* (page 405)
- *FieldSelectors* (page 426)

- *ImplicitPrelude* (page 295)
- *MonomorphismRestriction* (page 525)
- *NPlusKPatterns* (page 461)
- *NondecreasingIndentation* (page 733)
- *StarIsType* (page 378)
- *TraditionalRecordSyntax* (page 421)

Although not recommended, the deprecated *-fglasgow-exts* (page 273) flag enables a large swath of the extensions supported by GHC at once.

-fglasgow-exts

The flag `-fglasgow-exts` is equivalent to enabling the following extensions:

- *ConstrainedClassMethods* (page 471)
- *DeriveDataTypeable* (page 445)
- *DeriveFoldable* (page 442)
- *DeriveFunctor* (page 439)
- *DeriveGeneric* (page 598)
- *DeriveTraversable* (page 444)
- *EmptyDataDecls* (page 321)
- *ExistentialQuantification* (page 325)
- *ExplicitNamespaces* (page 319)
- *FlexibleContexts* (page 499)
- *FlexibleInstances* (page 480)
- *ForeignFunctionInterface* (page 561)
- *FunctionalDependencies* (page 475)
- *GeneralizedNewtypeDeriving* (page 447)
- *ImplicitParams* (page 517)
- *InterruptibleFFI* (page 565)
- *KindSignatures* (page 511)
- *LiberalTypeSynonyms* (page 324)
- *MagicHash* (page 278)
- *MultiParamTypeClasses* (page 470)
- *ParallelListComp* (page 288)
- *PatternGuards* (page 459)
- *PostfixOperators* (page 298)
- *RankNTypes* (page 402)
- *RecursiveDo* (page 279)
- *ScopedTypeVariables* (page 512)
- *StandaloneDeriving* (page 436)
- *TypeOperators* (page 323)
- *TypeSynonymInstances* (page 480)
- *UnboxedTuples* (page 556)
- *UnicodeSyntax* (page 277)
- *UnliftedFFITypes* (page 562)

Enabling these options is the *only* effect of `-fglasgow-exts`. We are trying to move away from this portmanteau flag, and towards enabling features individually.

6.1.2 Overview of all language extensions

GHC supports these language extensions:

Extension	Description
AllowAmbiguousTypes (page 509)	Allow the user to write ambiguous types, and the type inferen
ApplicativeDo (page 282)	Enable Applicative do-notation desugaring
Arrows (page 311)	Enable arrow notation extension
BangPatterns (page 541)	Enable bang patterns.
BinaryLiterals (page 491)	Enable support for binary literals.
BlockArguments (page 302)	Allow do blocks and other constructs as function arguments.
CApiFFI (page 566)	Enable the CAPI calling convention.
ConstrainedClassMethods (page 471)	Enable constrained class methods. Implied by *MultiParamTy*
ConstraintKinds (page 500)	Enable a kind of constraints.
CPP (page 240)	Enable the C preprocessor.
CUSKs (page 367)	Enable detection of complete user-supplied kind signatures.
DataKinds (page 357)	Enable datatype promotion.
DatatypeContexts (page 322)	Allow contexts on `data` types.
DeepSubsumption (page 405)	Enable deep subsumption
DefaultSignatures (page 472)	Enable default signatures.
DeriveAnyClass (page 453)	Enable deriving for any class.
DeriveDataTypeable (page 445)	Enable deriving for the `Data` class.
DeriveFoldable (page 442)	Enable deriving for the Foldable class. Implied by *DeriveTra*
DeriveFunctor (page 439)	Enable deriving for the Functor class. Implied by *DeriveTra*
DeriveGeneric (page 598)	Enable deriving for the Generic class.
DeriveLift (page 446)	Enable deriving for the Lift class
DeriveTraversable (page 444)	Enable deriving for the Traversable class. Implies *DeriveFun*
DerivingStrategies (page 455)	Enables deriving strategies.
DerivingVia (page 457)	Enable deriving instances `via` types of the same runtime rep
DisambiguateRecordFields (page 423)	Enable record field disambiguation. Implied by *RecordWildC*
DuplicateRecordFields (page 425)	Allow definition of record types with identically-named fields
EmptyCase (page 300)	Allow empty case alternatives.
EmptyDataDecls (page 321)	Allow definition of empty `data` types.
EmptyDataDeriving (page 435)	Allow deriving instances of standard type classes for empty d
ExistentialQuantification (page 325)	Enable liberalised type synonyms.
ExplicitForAll (page 506)	Enable explicit universal quantification. Implied by *ScopedTy*
ExplicitNamespaces (page 319)	Enable using the keyword `type` to specify the namespace of e
ExtendedDefaultRules (page 29)	Use GHCi's extended default rules in a normal module.
ExtendedLiterals (page 493)	Enable numeric literal postfix syntax for unboxed integers.
FieldSelectors (page 426)	Control visibility of field selector functions.
FlexibleContexts (page 499)	Remove some restrictions on class contexts
FlexibleInstances (page 480)	Enable flexible instances. Implies *TypeSynonymInstances* (pa
ForeignFunctionInterface (page 561)	Enable foreign function interface.
FunctionalDependencies (page 475)	Enable functional dependencies. Implies *MultiParamTypeCla*
GADTs (page 335)	Enable generalised algebraic data types. Implies *GADTSyntax*
GADTSyntax (page 329)	Enable generalised algebraic data type syntax.
GeneralisedNewtypeDeriving (page 447)	Enable newtype deriving.
GHC2021 (page 271)	Use GHC's set of default language extensions from 2021
GHC2024 (page 270)	Use GHC's set of default language extensions from 2024
GHCForeignImportPrim (page 565)	Enable prim calling convention. Intended for internal use onl
Haskell2010 (page 272)	Use the Haskell 2010 language edition.

Extension	Description
Haskell98 (page 272)	Use the Haskell 98 language edition.
HexFloatLiterals (page 492)	Enable support for *hexadecimal floating point literals* (page 4
ImplicitParams (page 517)	Enable Implicit Parameters.
ImplicitPrelude (page 295)	Don't implicitly `import Prelude`. Implied by *RebindableSyn*
ImportQualifiedPost (page 321)	`ImportQualifiedPost` allows the syntax `import M qualifie`
ImpredicativeTypes (page 407)	Enable impredicative types. Implies *RankNTypes* (page 402).
IncoherentInstances (page 486)	Enable incoherent instances. Implies *OverlappingInstances*
InstanceSigs (page 490)	Enable instance signatures.
InterruptibleFFI (page 565)	Enable interruptible FFI.
KindSignatures (page 511)	Enable kind signatures. Implied by *TypeFamilies* (page 338)
LambdaCase (page 299)	Enable lambda-case expressions.
LexicalNegation (page 317)	Use whitespace to determine whether the minus sign stands
LiberalTypeSynonyms (page 324)	Enable liberalised type synonyms.
LinearTypes (page 408)	Enable linear types. Implies *MonoLocalBinds* (page 526).
ListTuplePuns (page 361)	Enable punning for list, tuple and sum types.
MagicHash (page 278)	Allow `#` as a postfix modifier on identifiers.
MonadComprehensions (page 290)	Enable monad comprehensions.
MonoLocalBinds (page 526)	Enable do not generalise local bindings. Implied by *TypeFam*
MonomorphismRestriction (page 525)	Disable the monomorphism restriction.
MultiParamTypeClasses (page 470)	Enable multi parameter type classes. Implied by *Functional*
MultiWayIf (page 301)	Enable multi-way if-expressions.
NamedFieldPuns (page 427)	Enable record puns.
NamedWildCards (page 522)	Enable named wildcards.
NegativeLiterals (page 491)	Enable support for negative literals.
NondecreasingIndentation (page 733)	Allow nested contexts to be at the same indentation level as
NPlusKPatterns (page 461)	Enable support for `n+k` patterns. Implied by *Haskell98* (page
NullaryTypeClasses (page 475)	Deprecated, does nothing. nullary (no parameter) type class
NumDecimals (page 492)	Enable support for 'fractional' integer literals.
NumericUnderscores (page 494)	Enable support for *numeric underscores* (page 494).
OverlappingInstances (page 486)	Enable overlapping instances.
OverloadedLabels (page 496)	Enable overloaded labels.
OverloadedLists (page 293)	Enable overloaded lists.
OverloadedRecordDot (page 433)	Record '.' syntax
OverloadedRecordUpdate (page 433)	Record '.' syntax record updates
OverloadedStrings (page 495)	Enable overloaded string literals.
PackageImports (page 319)	Enable package-qualified imports.
ParallelListComp (page 288)	Enable parallel list comprehensions.
PartialTypeSignatures (page 520)	Enable partial type signatures.
PatternGuards (page 459)	Disable pattern guards. Implied by *Haskell98* (page 272).
PatternSynonyms (page 461)	Enable pattern synonyms.
PolyKinds (page 364)	Enable kind polymorphism. Implies *KindSignatures* (page 5
PostfixOperators (page 298)	Enable postfix operators.
QualifiedDo (page 284)	Enable qualified do-notation desugaring.
QuantifiedConstraints (page 502)	Allow `forall` quantifiers in constraints.
QuasiQuotes (page 538)	Enable quasiquotation.
Rank2Types (page 402)	Enable rank-2 types. Synonym for *RankNTypes* (page 402).
RankNTypes (page 402)	Enable rank-N types. Implied by *ImpredicativeTypes* (page
RebindableSyntax (page 295)	Employ rebindable syntax. Implies *NoImplicitPrelude* (page
RecordWildCards (page 428)	Enable record wildcards. Implies *DisambiguateRecordField*

Extension	Description
RecursiveDo (page 279)	Enable recursive do (mdo) notation.
RequiredTypeArguments (page 395)	Enable required type arguments in terms.
RoleAnnotations (page 419)	Enable role annotations.
Safe (page 586)	Enable the *Safe Haskell* (page 577) Safe mode.
ScopedTypeVariables (page 512)	Enable lexically-scoped type variables.
StandaloneDeriving (page 436)	Enable standalone deriving.
StandaloneKindSignatures (page 369)	Allow the use of standalone kind signatures.
StarIsType (page 378)	Treat * as `Data.Kind.Type`.
StaticPointers (page 552)	Enable static pointers.
Strict (page 543)	Make bindings in the current module strict by default.
StrictData (page 542)	Enable default strict datatype fields.
TemplateHaskell (page 528)	Enable Template Haskell.
TemplateHaskellQuotes (page 528)	Enable quotation subset of *Template Haskell* (page 527).
TraditionalRecordSyntax (page 421)	Disable support for traditional record syntax (as supported b
TransformListComp (page 288)	Enable generalised list comprehensions.
Trustworthy (page 586)	Enable the *Safe Haskell* (page 577) Trustworthy mode.
TupleSections (page 298)	Enable tuple sections.
TypeAbstractions (page 390)	Enable type abstraction syntax in patterns and type variable
TypeApplications (page 386)	Enable type application syntax in terms and types.
TypeData (page 363)	Enable type data declarations.
TypeFamilies (page 338)	Enable type families. Implies *ExplicitNamespaces* (page 31
TypeFamilyDependencies (page 355)	Enable injective type families. Implies *TypeFamilies* (page 3
TypeInType (page 364)	Deprecated. Enable kind polymorphism and datatype promot
TypeOperators (page 323)	Enable type operators. Implies *ExplicitNamespaces* (page 3
TypeSynonymInstances (page 480)	Enable type synonyms in instance heads. Implied by *Flexibl*
UnboxedSums (page 557)	Enable unboxed sums.
UnboxedTuples (page 556)	Enable the use of unboxed tuple syntax.
UndecidableInstances (page 484)	Enable undecidable instances.
UndecidableSuperClasses (page 470)	Allow all superclass constraints, including those that may res
UnicodeSyntax (page 277)	Enable unicode syntax.
UnliftedDatatypes (page 559)	Enable unlifted data types.
UnliftedFFITypes (page 562)	Enable unlifted FFI types
UnliftedNewtypes (page 558)	Enable unlifted newtypes.
Unsafe (page 587)	Enable *Safe Haskell* (page 577) Unsafe mode.
ViewPatterns (page 459)	Enable view patterns.

6.1.3 Summary of stolen syntax

Turning on an option that enables special syntax *might* cause working Haskell 98 code to fail to compile, perhaps because it uses a variable name which has become a reserved word. This section lists the syntax that is "stolen" by language extensions. We use notation and nonterminal names from the Haskell 98 lexical syntax (see the Haskell 98 Report). We only list syntax changes here that might affect existing working programs (i.e. "stolen" syntax). Many of these extensions will also enable new context-free syntax, but in all cases programs written to use the new syntax would not be compilable without the option enabled.

There are two classes of special syntax:

- New reserved words and symbols: character sequences which are no longer available for use as identifiers in the program.

- Other special syntax: sequences of characters that have a different meaning when this particular option is turned on.

The following syntax is stolen:

`forall` Stolen (in types) by default (see *Lexical syntax* (page 731)). `forall` is a reserved keyword and never a type variable, in accordance with GHC Proposal #43.

`mdo` Stolen by: *`RecursiveDo`* (page 279)

`foreign` Stolen by: *`ForeignFunctionInterface`* (page 561)

`rec`, `proc`, `-<`, `>-`, `-<<`, `>>-`, `(|`, `|)` Stolen by: *`Arrows`* (page 311)

`?varid` Stolen by: *`ImplicitParams`* (page 517)

`[|`, `[e|`, `[p|`, `[d|`, `[t|`, `[||`, `[e||` Stolen by: *`QuasiQuotes`* (page 538). Moreover, this introduces an ambiguity with list comprehension syntax. See the *discussion on quasi-quoting* (page 539) for details.

`$(`, `$$(`, `$varid`, `$$varid` Stolen by: *`TemplateHaskell`* (page 528)

`[varid|` Stolen by: *`QuasiQuotes`* (page 538)

⟨varid⟩, #⟨char⟩, #, ⟨string⟩, #, ⟨integer⟩, #, ⟨float⟩, #, ⟨float⟩, ## Stolen by: *`MagicHash`* (page 278)

⟨integer⟩, `#(Int|Word)(8|16|32|64)?` Stolen by: *`ExtendedLiterals`* (page 493)

`(#`, `#)` Stolen by: *`UnboxedTuples`* (page 556)

⟨varid⟩, `!`, ⟨varid⟩ Stolen by: *`BangPatterns`* (page 541)

`pattern` Stolen by: *`PatternSynonyms`* (page 461)

`static` Stolen by: *`StaticPointers`* (page 552)

6.2 Syntax

6.2.1 Unicode syntax

`UnicodeSyntax`

Since 6.8.1

Enable the use of Unicode characters in place of their equivalent ASCII sequences.

The language extension *`UnicodeSyntax`* (page 277) enables Unicode characters to be used to stand for certain ASCII character sequences. The following alternatives are provided:

ASCII	Unicode alternative	Code point	Name
`::`	∷	0x2237	PROPORTION
`=>`	⇒	0x21D2	RIGHTWARDS DOUBLE ARROW
`->`	→	0x2192	RIGHTWARDS ARROW
`<-`	←	0x2190	LEFTWARDS ARROW
`>-`	⤚	0x291a	RIGHTWARDS ARROW-TAIL
`-<`	⤙	0x2919	LEFTWARDS ARROW-TAIL
`>>-`	⤜	0x291C	RIGHTWARDS DOUBLE ARROW-TAIL
`-<<`	⤛	0x291B	LEFTWARDS DOUBLE ARROW-TAIL
`*`	★	0x2605	BLACK STAR
`forall`	∀	0x2200	FOR ALL
`(|`	⦇	0x2987	Z NOTATION LEFT IMAGE BRACKET
`|)`	⦈	0x2988	Z NOTATION RIGHT IMAGE BRACKET
`[|`	⟦	0x27E6	MATHEMATICAL LEFT WHITE SQUARE BRACKET
`|]`	⟧	0x27E7	MATHEMATICAL RIGHT WHITE SQUARE BRACKET
`%1->`	⊸	0x22B8	MULTIMAP

6.2.2 The magic hash

MagicHash

Since 6.8.1

Enables the use of the hash character (#) as an identifier suffix.

The language extension *MagicHash* (page 278) allows `#` as a postfix modifier to identifiers. Thus, `x#` is a valid variable, and `T#` is a valid type constructor or data constructor.

The hash sign does not change semantics at all. We tend to use variable names ending in "#" for unboxed values or types (e.g. `Int#`), but there is no requirement to do so; they are just plain ordinary variables. Nor does the *MagicHash* (page 278) extension bring anything into scope. For example, to bring `Int#` into scope you must import `GHC.Exts` (see *Unboxed types and primitive operations* (page 554)); the *MagicHash* (page 278) extension then allows you to *refer* to the `Int#` that is now in scope. Note that with this option, the meaning of `x#y = 0` is changed: it defines a function `x#` taking a single argument `y`; to define the operator `#`, put a space: `x # y = 0`.

The *MagicHash* (page 278) also enables some new forms of literals (see *Unboxed types* (page 554)):

- `'x'#` has type `Char#`
- `"foo"#` has type `Addr#`
- `3#` has type `Int#`. In general, any Haskell integer lexeme followed by a `#` is an `Int#` literal, e.g. `-0x3A#` as well as `32#`.
- `3##` has type `Word#`. In general, any non-negative Haskell integer lexeme followed by `##` is a `Word#`.
- `3.2#` has type `Float#`.
- `3.2##` has type `Double#`

6.2.3 The recursive do-notation

RecursiveDo

Since 6.8.1

Allow the use of recursive `do` notation.

The do-notation of Haskell 98 does not allow *recursive bindings*, that is, the variables bound in a do-expression are visible only in the textually following code block. Compare this to a let-expression, where bound variables are visible in the entire binding group.

It turns out that such recursive bindings do indeed make sense for a variety of monads, but not all. In particular, recursion in this sense requires a fixed-point operator for the underlying monad, captured by the `mfix` method of the `MonadFix` class, defined in `Control.Monad.Fix` as follows:

```
class Monad m => MonadFix m where
   mfix :: (a -> m a) -> m a
```

Haskell's `Maybe`, `[]` (list), `ST` (both strict and lazy versions), `IO`, and many other monads have `MonadFix` instances. On the negative side, the continuation monad, with the signature `(a -> r) -> r`, does not.

For monads that do belong to the `MonadFix` class, GHC provides an extended version of the do-notation that allows recursive bindings. The *RecursiveDo* (page 279) (language pragma: `RecursiveDo`) provides the necessary syntactic support, introducing the keywords `mdo` and `rec` for higher and lower levels of the notation respectively. Unlike bindings in a `do` expression, those introduced by `mdo` and `rec` are recursively defined, much like in an ordinary let-expression. Due to the new keyword `mdo`, we also call this notation the *mdo-notation*.

Here is a simple (albeit contrived) example:

```
{-# LANGUAGE RecursiveDo #-}
justOnes = mdo { xs <- Just (1:xs)
               ; return (map negate xs) }
```

or equivalently

```
{-# LANGUAGE RecursiveDo #-}
justOnes = do { rec { xs <- Just (1:xs) }
              ; return (map negate xs) }
```

As you can guess `justOnes` will evaluate to `Just [-1,-1,-1,...`.

GHC's implementation the mdo-notation closely follows the original translation as described in the paper A recursive do for Haskell, which in turn is based on the work Value Recursion in Monadic Computations. Furthermore, GHC extends the syntax described in the former paper with a lower level syntax flagged by the `rec` keyword, as we describe next.

6.2.3.1 Recursive binding groups

The extension *RecursiveDo* (page 279) also introduces a new keyword `rec`, which wraps a mutually-recursive group of monadic statements inside a `do` expression, producing a single statement. Similar to a `let` statement inside a `do`, variables bound in the `rec` are visible throughout the `rec` group, and below it. For example, compare

```
do { a <- getChar              do { a <- getChar
   ; let { r1 = f a r2            ; rec { r1 <- f a r2
   ;     ; r2 = g r1 }            ;     ; r2 <- g r1 }
   ; return (r1 ++ r2) }          ; return (r1 ++ r2) }
```

In both cases, `r1` and `r2` are available both throughout the `let` or `rec` block, and in the statements that follow it. The difference is that `let` is non-monadic, while `rec` is monadic. (In Haskell `let` is really `letrec`, of course.)

The semantics of `rec` is fairly straightforward. Whenever GHC finds a `rec` group, it will compute its set of bound variables, and will introduce an appropriate call to the underlying monadic value-recursion operator `mfix`, belonging to the `MonadFix` class. Here is an example:

```
rec { b <- f a c     ===>    (b,c) <- mfix (\ ~(b,c) -> do { b <- f a c
    ; c <- f b a }                                          ; c <- f b a
                                                            ; return (b,c) })
```

As usual, the meta-variables `b`, `c` etc., can be arbitrary patterns. In general, the statement `rec ss` is desugared to the statement

```
vs <- mfix (\ ~vs -> do { ss; return vs })
```

where `vs` is a tuple of the variables bound by `ss`.

Note in particular that the translation for a `rec` block only involves wrapping a call to `mfix`: it performs no other analysis on the bindings. The latter is the task for the `mdo` notation, which is described next.

6.2.3.2 The `mdo` notation

A `rec`-block tells the compiler where precisely the recursive knot should be tied. It turns out that the placement of the recursive knots can be rather delicate: in particular, we would like the knots to be wrapped around as minimal groups as possible. This process is known as *segmentation*, and is described in detail in Section 3.2 of A recursive do for Haskell. Segmentation improves polymorphism and reduces the size of the recursive knot. Most importantly, it avoids unnecessary interference caused by a fundamental issue with the so-called *right-shrinking* axiom for monadic recursion. In brief, most monads of interest (IO, strict state, etc.) do *not* have recursion operators that satisfy this axiom, and thus not performing segmentation can cause unnecessary interference, changing the termination behavior of the resulting translation. (Details can be found in Sections 3.1 and 7.2.2 of Value Recursion in Monadic Computations.)

The `mdo` notation removes the burden of placing explicit `rec` blocks in the code. Unlike an ordinary `do` expression, in which variables bound by statements are only in scope for later statements, variables bound in an `mdo` expression are in scope for all statements of the expression. The compiler then automatically identifies minimal mutually recursively dependent segments of statements, treating them as if the user had wrapped a `rec` qualifier around them.

The definition is syntactic:

- A generator ⟨g⟩ *depends* on a textually following generator ⟨g'⟩, if
  - ⟨g'⟩ defines a variable that is used by ⟨g⟩, or
  - ⟨g'⟩ textually appears between ⟨g⟩ and ⟨g''⟩, where ⟨g⟩ depends on ⟨g''⟩.
- A *segment* of a given mdo-expression is a minimal sequence of generators such that no generator of the sequence depends on an outside generator. As a special case, although it is not a generator, the final expression in an mdo-expression is considered to form a segment by itself.

Segments in this sense are related to *strongly-connected components* analysis, with the exception that bindings in a segment cannot be reordered and must be contiguous.

Here is an example mdo-expression, and its translation to rec blocks:

```
mdo { a <- getChar      ===> do { a <- getChar
    ; b <- f a c                ; rec { b <- f a c
    ; c <- f b a                ;     ; c <- f b a }
    ; z <- h a b                ; z <- h a b
    ; d <- g d e                ; rec { d <- g d e
    ; e <- g a z                ;     ; e <- g a z }
    ; putChar c }               ; putChar c }
```

Note that a given mdo expression can cause the creation of multiple rec blocks. If there are no recursive dependencies, mdo will introduce no rec blocks. In this latter case an mdo expression is precisely the same as a do expression, as one would expect.

In summary, given an mdo expression, GHC first performs segmentation, introducing rec blocks to wrap over minimal recursive groups. Then, each resulting rec is desugared, using a call to Control.Monad.Fix.mfix as described in the previous section. The original mdo-expression typechecks exactly when the desugared version would do so.

Here are some other important points in using the recursive-do notation:

- It is enabled with the extension RecursiveDo (page 279), or the LANGUAGE RecursiveDo pragma. (The same extension enables both mdo-notation, and the use of rec blocks inside do expressions.)
- rec blocks can also be used inside mdo-expressions, which will be treated as a single statement. However, it is good style to either use mdo or rec blocks in a single expression.
- If recursive bindings are required for a monad, then that monad must be declared an instance of the MonadFix class.
- The following instances of MonadFix are automatically provided: List, Maybe, IO. Furthermore, the Control.Monad.ST and Control.Monad.ST.Lazy modules provide the instances of the MonadFix class for Haskell's internal state monad (strict and lazy, respectively).
- Like let and where bindings, name shadowing is not allowed within an mdo-expression or a rec-block; that is, all the names bound in a single rec must be distinct. (GHC will complain if this is not the case.)

6.2.4 Applicative do-notation

ApplicativeDo

Since 8.0.1

Allow use of Applicative do notation.

The language option *ApplicativeDo* (page 282) enables an alternative translation for the do-notation, which uses the operators <$>, <*>, along with join as far as possible. There are two main reasons for wanting to do this:

- We can use do-notation with types that are an instance of Applicative and Functor, but not Monad
- In some monads, using the applicative operators is more efficient than monadic bind. For example, it may enable more parallelism.

Applicative do-notation desugaring preserves the original semantics, provided that the Applicative instance satisfies <*> = ap and pure = return (these are true of all the common monadic types). Thus, you can normally turn on *ApplicativeDo* (page 282) without fear of breaking your program. There is one pitfall to watch out for; see *Things to watch out for* (page 284).

There are no syntactic changes with *ApplicativeDo* (page 282). The only way it shows up at the source level is that you can have a do expression that doesn't require a Monad constraint. For example, in GHCi:

```
Prelude> :set -XApplicativeDo
Prelude> :t \m -> do { x <- m; return (not x) }
\m -> do { x <- m; return (not x) }
  :: Functor f => f Bool -> f Bool
```

This example only requires Functor, because it is translated into (\x -> not x) <$> m. A more complex example requires Applicative,

```
Prelude> :t \m -> do { x <- m 'a'; y <- m 'b'; return (x || y) }
\m -> do { x <- m 'a'; y <- m 'b'; return (x || y) }
  :: Applicative f => (Char -> f Bool) -> f Bool
```

Here GHC has translated the expression into

```
(\x y -> x || y) <$> m 'a' <*> m 'b'
```

It is possible to see the actual translation by using *-ddump-ds* (page 258), but be warned, the output is quite verbose.

Note that if the expression can't be translated into uses of <$>, <*> only, then it will incur a Monad constraint as usual. This happens when there is a dependency on a value produced by an earlier statement in the do-block:

```
Prelude> :t \m -> do { x <- m True; y <- m x; return (x || y) }
\m -> do { x <- m True; y <- m x; return (x || y) }
  :: Monad m => (Bool -> m Bool) -> m Bool
```

Here, m x depends on the value of x produced by the first statement, so the expression cannot be translated using <*>.

In general, the rule for when a do statement incurs a Monad constraint is as follows. If the do-expression has the following form:

```
do p1 <- E1; ...; pn <- En; return E
```

where none of the variables defined by `p1...pn` are mentioned in `E1...En`, and `p1...pn` are all variables or lazy patterns, then the expression will only require `Applicative`. The do expression may also contain `let` statements anywhere, provided that the right-hand-sides of the `let` bindings do not mention any of `p1...pn`. Otherwise, the expression will require `Monad`. The block may return a pure expression `E` depending upon the results `p1...pn` and the `let` bindings, with either `return` or `pure`.

Note: the final statement must match one of these patterns exactly:

- `return E`
- `return $ E`
- `pure E`
- `pure $ E`

otherwise GHC cannot recognise it as a `return` statement, and the transformation to use <$> that we saw above does not apply. In particular, slight variations such as `return . Just $ x` or `let x = e in return x` would not be recognised.

If the final statement is not of one of these forms, GHC falls back to standard `do` desugaring, and the expression will require a `Monad` constraint.

When the statements of a `do` expression have dependencies between them, and `ApplicativeDo` cannot infer an `Applicative` type, it uses a heuristic algorithm to try to use <*> as much as possible. This algorithm usually finds the best solution, but in rare complex cases it might miss an opportunity. There is an algorithm that finds the optimal solution, provided as an option:

`-foptimal-applicative-do`

Since 8.0.1

Enables an alternative algorithm for choosing where to use <*> in conjunction with the `ApplicativeDo` language extension. This algorithm always finds the optimal solution, but it is expensive: `O(n^3)`, so this option can lead to long compile times when there are very large `do` expressions (over 100 statements). The default `ApplicativeDo` algorithm is `O(n^2)`.

6.2.4.1 Strict patterns

A strict pattern match in a bind statement prevents `ApplicativeDo` from transforming that statement to use `Applicative`. This is because the transformation would change the semantics by making the expression lazier.

For example, this code will require a `Monad` constraint:

```
> :t \m -> do { (x:xs) <- m; return x }
\m -> do { (x:xs) <- m; return x } :: Monad m => m [b] -> m b
```

but making the pattern match lazy allows it to have a `Functor` constraint:

```
> :t \m -> do { ~(x:xs) <- m; return x }
\m -> do { ~(x:xs) <- m; return x } :: Functor f => f [b] -> f b
```

A "strict pattern match" is any pattern match that can fail. For example, `()`, `(x:xs)`, `!z`, and `C x` are strict patterns, but `x` and `~(1,2)` are not. For the purposes of `ApplicativeDo`, a pattern match against a `newtype` constructor is considered strict.

When there's a strict pattern match in a sequence of statements, `ApplicativeDo` places a `>>=` between that statement and the one that follows it. The sequence may be transformed to use `<*>` elsewhere, but the strict pattern match and the following statement will always be connected with `>>=`, to retain the same strictness semantics as the standard do-notation. If you don't want this, simply put a `~` on the pattern match to make it lazy.

6.2.4.2 Things to watch out for

Your code should just work as before when *ApplicativeDo* (page 282) is enabled, provided you use conventional `Applicative` instances. However, if you define a `Functor` or `Applicative` instance using do-notation, then it will likely get turned into an infinite loop by GHC. For example, if you do this:

```
instance Functor MyType where
    fmap f m = do x <- m; return (f x)
```

Then applicative desugaring will turn it into

```
instance Functor MyType where
    fmap f m = fmap (\x -> f x) m
```

And the program will loop at runtime. Similarly, an `Applicative` instance like this

```
instance Applicative MyType where
    pure = return
    x <*> y = do f <- x; a <- y; return (f a)
```

will result in an infinite loop when `<*>` is called.

Just as you wouldn't define a `Monad` instance using the do-notation, you shouldn't define `Functor` or `Applicative` instance using do-notation (when using `ApplicativeDo`) either. The correct way to define these instances in terms of `Monad` is to use the `Monad` operations directly, e.g.

```
instance Functor MyType where
    fmap f m = m >>= return . f

instance Applicative MyType where
    pure = return
    (<*>) = ap
```

6.2.5 Qualified do-notation

`QualifiedDo`

Since 9.0.1

Allow the use of qualified `do` notation.

`QualifiedDo` enables qualifying a `do` block with a module name, to control which operations to use for the monadic combinators that the `do` notation desugars to. When `-XQualifiedDo` is enabled, you can *qualify* the do notation by writing `modid.do`, where `modid` is a module name in scope:

```
{-# LANGUAGE QualifiedDo #-}
import qualified Some.Module.Monad as M

action :: M.SomeType a
action = M.do x <- u
              res
              M.return x
```

The additional module name (here `M`) is called the qualifier of the do-expression.

The unqualified `do` syntax is convenient for writing monadic code, but it only works for data types that provide an instance of the `Monad` type class. There are other types which are "monad-like" but can't provide an instance of `Monad` (e.g. indexed monads, graded monads or relative monads), yet they could still use the `do` syntax if it weren't hardwired to the methods of the `Monad` type class. `-XQualifiedDo` comes to make the do syntax customizable in this respect. It allows you to mix and match `do` blocks of different types with suitable operations to use on each case:

```
{-# LANGUAGE QualifiedDo #-}
import qualified Control.Monad.Linear as L

import MAC (label, box, runMAC)
import qualified MAC as MAC

f :: IO ()
f = do
  x <- runMAC $           -- (Prelude.>>=)
                          --   (runMAC $
    MAC.do                --
      d <- label "y"      --     label "y" MAC.>>= \d ->
      box $               --
                          --       (box $
        L.do              --
          r <- L.f d      --         L.f d L.>>= \r ->
          L.g r           --         L.g r L.>>
          L.return r      --         L.return r
                          --       ) MAC.>>
      MAC.return d        --       (MAC.return d)
                          --   )
  print x                 --   (\x -> print x)
```

The semantics of `do` notation statements with `-XQualifiedDo` is as follows:

- The `x <- u` statement uses `(M.>>=)`

```
M.do { x <- u; stmts }  =  u M.>>= \x -> M.do { stmts }
```

- The `u` statement uses `(M.>>)`

```
M.do { u; stmts }  =  u M.>> M.do { stmts }
```

- The a `pat <- u` statement uses `M.fail` for the failing case, if such a case is needed

```
M.do { pat <- u; stmts }  =  u M.>>= \case
  { pat -> M.do { stmts }
```

(continues on next page)

(continued from previous page)

```
  ; _ -> M.fail "…"
  }
```

If the pattern cannot fail, then we don't need to use `M.fail`.

```
M.do { pat <- u; stmts }  =  u M.>>= \case pat -> M.do { stmts }
```

- The desugaring of `-XApplicativeDo` uses `M.fmap`, `(M.<*>)`, and `M.join` (after the applicative-do grouping has been performed)

```
M.do { (x1 <- u1 | … | xn <- un); M.return e }  =
  (\x1 … xn -> e) `M.fmap` u1 M.<*> … M.<*> un

M.do { (x1 <- u1 | … | xn <- un); stmts }  =
  M.join ((\x1 … xn -> M.do { stmts }) `M.fmap` u1 M.<*> … M.<*> un)
```

Note that `M.join` is only needed if the final expression is not identifiably a `return`. With `-XQualifiedDo` enabled, `-XApplicativeDo` looks only for the qualified `return`/`pure` in a qualified do-block.

- With `-XRecursiveDo`, `rec` and `mdo` blocks use `M.mfix` and `M.return`:

```
M.do { rec { x1 <- u1; … ; xn <- un }; stmts }  =
  M.do
  { (x1, …, xn) <- M.mfix (\~(x1, …, xn) -> M.do { x1 <- u1; …; xn <- un; M.
↪return (x1, …, xn)})
  ; stmts
  }
```

If a name `M.op` is required by the desugaring process (and only if it's required!) but the name is not in scope, it is reported as an error.

The types of the operations picked for desugaring must produce an expression which is accepted by the typechecker. But other than that, there are no specific requirements on the types.

If no qualifier is specified with `-XQualifiedDo` enabled, it defaults to the operations defined in the Prelude, or, if `-XRebindableSyntax` is enabled, to whatever operations are in scope.

Note that the operations to be qualified must be in scope for QualifiedDo to work. I.e. `import MAC (label)` in the example above would result in an error, since `MAC.>>=` and `MAC.>>` would not be in scope.

6.2.5.1 Examples

`-XQualifiedDo` does not affect `return` in the monadic `do` notation.

```
import qualified Some.Monad.M as M

boolM :: (a -> M.M Bool) -> b -> b -> a -> M.M b
boolM p a b x = M.do
    px <- p x     -- M.>>=
    if px then
      return b    -- Prelude.return
    else
      M.return a  -- M.return
```

`-XQualifiedDo` does not affect explicit `(>>=)` in the monadic `do` notation.

```
import qualified Some.Monad.M as M
import Data.Bool (bool)

boolMM :: (a -> M.M Bool) -> M b -> M b -> a -> M.M b
boolMM p ma mb x = M.do
    p x >>= bool ma mb   -- Prelude.>>=
```

Nested `do` blocks do not affect each other's meanings.

```
import qualified Some.Monad.M as M

f :: M.M SomeType
f = M.do
    x <- f1                 -- M.>>=
    f2 (do y <- g1          -- Prelude.>>=
           g2 x y)
  where
    f1 = ...
    f2 m = ...
    g1 = ...
    g2 x y = ...
```

The type of `(>>=)` can also be modified, as seen here for a graded monad:

```
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE TypeFamilies #-}
module Control.Monad.Graded (GradedMonad(..)) where

import Data.Kind (Constraint)

class GradedMonad (m :: k -> * -> *) where
  type Unit m :: k
  type Plus m (i :: k) (j :: k) :: k
  type Inv  m (i :: k) (j :: k) :: Constraint
  (>>=) :: Inv m i j => m i a -> (a -> m j b) -> m (Plus m i j) b
  return :: a -> m (Unit m) a

-----------------

module M where

import Control.Monad.Graded as Graded

g :: GradedMonad m => a -> m SomeTypeIndex b
g a = Graded.do
  b <- someGradedFunction a Graded.>>= someOtherGradedFunction
  c <- anotherGradedFunction b
  Graded.return c
```

6.2.6 Parallel List Comprehensions

ParallelListComp

Since 6.8.1

Allow parallel list comprehension syntax.

Parallel list comprehensions are a natural extension to list comprehensions. List comprehensions can be thought of as a nice syntax for writing maps and filters. Parallel comprehensions extend this to include the `zipWith` family.

A parallel list comprehension has multiple independent branches of qualifier lists, each separated by a `|` symbol. For example, the following zips together two lists:

```
[ (x, y) | x <- xs | y <- ys ]
```

The behaviour of parallel list comprehensions follows that of zip, in that the resulting list will have the same length as the shortest branch.

We can define parallel list comprehensions by translation to regular comprehensions. Here's the basic idea:

Given a parallel comprehension of the form:

```
[ e | p1 <- e11, p2 <- e12, ...
    | q1 <- e21, q2 <- e22, ...
    ...
]
```

This will be translated to:

```
[ e | ((p1,p2), (q1,q2), ...) <- zipN [(p1,p2) | p1 <- e11, p2 <- e12, ...]
                                       [(q1,q2) | q1 <- e21, q2 <- e22, ...]
                                       ...
]
```

where `zipN` is the appropriate zip for the given number of branches.

6.2.7 Generalised (SQL-like) List Comprehensions

TransformListComp

Since 6.10.1

Allow use of generalised list (SQL-like) comprehension syntax. This introduces the `group`, `by`, and `using` keywords.

Generalised list comprehensions are a further enhancement to the list comprehension syntactic sugar to allow operations such as sorting and grouping which are familiar from SQL. They are fully described in the paper Comprehensive comprehensions: comprehensions with "order by" and "group by", except that the syntax we use differs slightly from the paper.

The extension is enabled with the extension *TransformListComp* (page 288).

Here is an example:

```
employees = [ ("Simon", "MS", 80)
            , ("Erik", "MS", 100)
            , ("Phil", "Ed", 40)
            , ("Gordon", "Ed", 45)
            , ("Paul", "Yale", 60) ]

output = [ (the dept, sum salary)
         | (name, dept, salary) <- employees
         , then group by dept using groupWith
         , then sortWith by (sum salary)
         , then take 5 ]
```

In this example, the list `output` would take on the value:

```
[("Yale", 60), ("Ed", 85), ("MS", 180)]
```

There are three new keywords: `group`, `by`, and `using`. (The functions `sortWith` and `groupWith` are not keywords; they are ordinary functions that are exported by `GHC.Exts`.)

There are five new forms of comprehension qualifier, all introduced by the (existing) keyword `then`:

- ```
  then f
  ```

  This statement requires that f have the type forall a. [a] -> [a] . You can see an example of its use in the motivating example, as this form is used to apply take 5 .

- ```
  then f by e
  ```

  This form is similar to the previous one, but allows you to create a function which will be passed as the first argument to f. As a consequence f must have the type `forall a. (a -> t) -> [a] -> [a]`. As you can see from the type, this function lets f "project out" some information from the elements of the list it is transforming.

  An example is shown in the opening example, where `sortWith` is supplied with a function that lets it find out the `sum salary` for any item in the list comprehension it transforms.

- ```
  then group by e using f
  ```

  This is the most general of the grouping-type statements. In this form, f is required to have type `forall a. (a -> t) -> [a] -> [[a]]`. As with the `then f by e` case above, the first argument is a function supplied to f by the compiler which lets it compute e on every element of the list being transformed. However, unlike the non-grouping case, f additionally partitions the list into a number of sublists: this means that at every point after this statement, binders occurring before it in the comprehension refer to *lists* of possible values, not single values. To help understand this, let's look at an example:

  ```
  -- This works similarly to groupWith in GHC.Exts, but doesn't sort its input first
  groupRuns :: Eq b => (a -> b) -> [a] -> [[a]]
  groupRuns f = groupBy (\x y -> f x == f y)

  output = [ (the x, y)
  | x <- ([1..3] ++ [1..2])
  , y <- [4..6]
  , then group by x using groupRuns ]
  ```

  This results in the variable `output` taking on the value below:
  ```

```
[(1, [4, 5, 6]), (2, [4, 5, 6]), (3, [4, 5, 6]), (1, [4, 5, 6]), (2, [4, 5, 6])]
```

Note that we have used the `the` function to change the type of x from a list to its original numeric type. The variable y, in contrast, is left unchanged from the list form introduced by the grouping.

- ```
  then group using f
  ```

  With this form of the group statement, f is required to simply have the type `forall a. [a] -> [[a]]`, which will be used to group up the comprehension so far directly. An example of this form is as follows:

  ```
  output = [ x
  | y <- [1..5]
  , x <- "hello"
  , then group using inits]
  ```

  This will yield a list containing every prefix of the word "hello" written out 5 times:

  ```
  ["","h","he","hel","hell","hello","helloh","hellohe","hellohel","hellohell",
  ↪"hellohello","hellohelloh",...]
  ```

6.2.8 Monad comprehensions

MonadComprehensions

Since 7.2.1

Enable list comprehension syntax for arbitrary monads.

Monad comprehensions generalise the list comprehension notation, including parallel comprehensions (*Parallel List Comprehensions* (page 288)) and transform comprehensions (*Generalised (SQL-like) List Comprehensions* (page 288)) to work for any monad.

Monad comprehensions support:

- Bindings:

  ```
  [ x + y | x <- Just 1, y <- Just 2 ]
  ```

  Bindings are translated with the `(>>=)` and `return` functions to the usual do-notation:

  ```
  do x <- Just 1
     y <- Just 2
     return (x+y)
  ```

- Guards:

  ```
  [ x | x <- [1..10], x <= 5 ]
  ```

  Guards are translated with the `guard` function, which requires an `Alternative` instance:

  ```
  do x <- [1..10]
     guard (x <= 5)
     return x
  ```

- Transform statements (as with *TransformListComp* (page 288)):
  ```

```
[ x+y | x <- [1..10], y <- [1..x], then take 2 ]
```

This translates to:

```
do (x,y) <- take 2 (do x <- [1..10]
                       y <- [1..x]
                       return (x,y))
   return (x+y)
```

- Group statements (as with *TransformListComp* (page 288)):

```
[ x | x <- [1,1,2,2,3], then group by x using GHC.Exts.groupWith ]
[ x | x <- [1,1,2,2,3], then group using myGroup ]
```

- Parallel statements (as with *ParallelListComp* (page 288)):

```
[ (x+y) | x <- [1..10]
        | y <- [11..20]
        ]
```

Parallel statements are translated using the `mzip` function, which requires a `MonadZip` instance defined in Control.Monad.Zip:

```
do (x,y) <- mzip (do x <- [1..10]
                     return x)
                 (do y <- [11..20]
                     return y)
   return (x+y)
```

All these features are enabled by default if the *MonadComprehensions* (page 290) extension is enabled. The types and more detailed examples on how to use comprehensions are explained in the previous chapters *Generalised (SQL-like) List Comprehensions* (page 288) and *Parallel List Comprehensions* (page 288). In general you just have to replace the type `[a]` with the type `Monad m => m a` for monad comprehensions.

Note: Even though most of these examples are using the list monad, monad comprehensions work for any monad. The `base` package offers all necessary instances for lists, which make *MonadComprehensions* (page 290) backward compatible to built-in, transform and parallel list comprehensions.

More formally, the desugaring is as follows. We write `D[ e | Q]` to mean the desugaring of the monad comprehension `[ e | Q]`:

```
Expressions: e
Declarations: d
Lists of qualifiers: Q,R,S

-- Basic forms
D[ e | ]               = return e
D[ e | p <- e, Q ]  = e >>= \p -> D[ e | Q ]
D[ e | e, Q ]          = guard e >> \p -> D[ e | Q ]
D[ e | let d, Q ]      = let d in D[ e | Q ]

-- Parallel comprehensions (iterate for multiple parallel branches)
D[ e | (Q | R), S ]    = mzip D[ Qv | Q ] D[ Rv | R ] >>= \(Qv,Rv) -> D[ e | S ]
```

(continues on next page)

(continued from previous page)

```
-- Transform comprehensions
D[ e | Q then f, R ]                  = f D[ Qv | Q ] >>= \Qv -> D[ e | R ]

D[ e | Q then f by b, R ]             = f (\Qv -> b) D[ Qv | Q ] >>= \Qv -> D[ e | R ]

D[ e | Q then group using f, R ]      = f D[ Qv | Q ] >>= \ys ->
                                        case (fmap selQv1 ys, ..., fmap selQvn ys) of
                                         Qv -> D[ e | R ]

D[ e | Q then group by b using f, R ] = f (\Qv -> b) D[ Qv | Q ] >>= \ys ->
                                        case (fmap selQv1 ys, ..., fmap selQvn ys) of
                                           Qv -> D[ e | R ]

where  Qv is the tuple of variables bound by Q (and used subsequently)
       selQvi is a selector mapping Qv to the ith component of Qv

Operator     Standard binding       Expected type
--------------------------------------------------------------------
return       GHC.Base               t1 -> m t2
(>>=)        GHC.Base               m1 t1 -> (t2 -> m2 t3) -> m3 t3
(>>)         GHC.Base               m1 t1 -> m2 t2         -> m3 t3
guard        Control.Monad          t1 -> m t2
fmap         GHC.Base               forall a b. (a->b) -> n a -> n b
mzip         Control.Monad.Zip      forall a b. m a -> m b -> m (a,b)
```

The comprehension should typecheck when its desugaring would typecheck, except that (as discussed in *Generalised (SQL-like) List Comprehensions* (page 288)) in the "then `f`" and "then group using `f`" clauses, when the "by `b`" qualifier is omitted, argument `f` should have a polymorphic type. In particular, "then `Data.List.sort`" and "then group using `Data.List.group`" are insufficiently polymorphic.

Monad comprehensions support rebindable syntax (*Rebindable syntax and the implicit Prelude import* (page 295)). Without rebindable syntax, the operators from the "standard binding" module are used; with rebindable syntax, the operators are looked up in the current lexical scope. For example, parallel comprehensions will be typechecked and desugared using whatever "`mzip`" is in scope.

The rebindable operators must have the "Expected type" given in the table above. These types are surprisingly general. For example, you can use a bind operator with the type

```
(>>=) :: T x y a -> (a -> T y z b) -> T x z b
```

In the case of transform comprehensions, notice that the groups are parameterised over some arbitrary type `n` (provided it has an `fmap`, as well as the comprehension being over an arbitrary monad.

6.2.9 Overloaded lists

OverloadedLists

Since 7.8.1

Enable overloaded list syntax (e.g. desugaring of lists via the `IsList` class).

GHC supports *overloading of the list notation*. Let us recap the notation for constructing lists. In Haskell, the list notation can be used in the following seven ways:

```
[]          -- Empty list
[x]         -- x : []
[x,y,z]     -- x : y : z : []
[x .. ]     -- enumFrom x
[x,y ..]    -- enumFromThen x y
[x .. y]    -- enumFromTo x y
[x,y .. z]  -- enumFromThenTo x y z
```

When the `OverloadedLists` extension is turned on, the aforementioned seven notations are desugared as follows:

```
[]          -- fromListN 0 []
[x]         -- fromListN 1 (x : [])
[x,y,z]     -- fromListN 3 (x : y : z : [])
[x .. ]     -- fromList (enumFrom x)
[x,y ..]    -- fromList (enumFromThen x y)
[x .. y]    -- fromList (enumFromTo x y)
[x,y .. z]  -- fromList (enumFromThenTo x y z)
```

This extension allows programmers to use the list notation for construction of structures like: `Set`, `Map`, `IntMap`, `Vector`, `Text` and `Array`. The following code listing gives a few examples:

```
['0' .. '9']             :: Set Char
[1 .. 10]                :: Vector Int
[("default",0), (k1,v1)] :: Map String Int
['a' .. 'z']             :: Text
```

List patterns are also overloaded. When the `OverloadedLists` extension is turned on, these definitions are desugared as follows

```
f [] = ...          -- f (toList -> []) = ...
g [x,y,z] = ...     -- g (toList -> [x,y,z]) = ...
```

(Here we are using view-pattern syntax for the translation, see *View patterns* (page 459).)

6.2.9.1 The IsList class

In the above desugarings, the functions `toList`, `fromList` and `fromListN` are all methods of the `IsList` class, which is itself exported from the `GHC.Exts` module. The type class is defined as follows:

```
class IsList l where
  type Item l

  fromList :: [Item l] -> l
  toList   :: l -> [Item l]
```

(continues on next page)

(continued from previous page)

```
  fromListN :: Int -> [Item l] -> l
  fromListN _ = fromList
```

The `IsList` class and its methods are intended to be used in conjunction with the `OverloadedLists` extension.

- The type function `Item` returns the type of items of the structure `l`.
- The function `fromList` constructs the structure `l` from the given list of `Item l`.
- The function `fromListN` takes the input list's length as a hint. Its behaviour should be equivalent to `fromList`. The hint can be used for more efficient construction of the structure `l` compared to `fromList`. If the given hint is not equal to the input list's length the behaviour of `fromListN` is not specified.
- The function `toList` should be the inverse of `fromList`.

It is perfectly fine to declare new instances of `IsList`, so that list notation becomes useful for completely new data types. Here are several example instances:

```
instance IsList [a] where
  type Item [a] = a
  fromList = id
  toList = id

instance (Ord a) => IsList (Set a) where
  type Item (Set a) = a
  fromList = Set.fromList
  toList = Set.toList

instance (Ord k) => IsList (Map k v) where
  type Item (Map k v) = (k,v)
  fromList = Map.fromList
  toList = Map.toList

instance IsList (IntMap v) where
  type Item (IntMap v) = (Int,v)
  fromList = IntMap.fromList
  toList = IntMap.toList

instance IsList Text where
  type Item Text = Char
  fromList = Text.pack
  toList = Text.unpack

instance IsList (Vector a) where
  type Item (Vector a) = a
  fromList  = Vector.fromList
  fromListN = Vector.fromListN
  toList = Vector.toList
```

Users should not, however, provide any instance that overlaps with the first instance above. Namely, `fromList` and `toList`, when used on lists, should always be the identity function. For example, the following instance is disallowed:

```
instance {-# OVERLAPPING #-} IsList [Int] where
  type Item [Int] = Int
```

(continues on next page)

(continued from previous page)

```
fromList = reverse
toList = reverse
```

6.2.9.2 Rebindable syntax

When desugaring list notation with *OverloadedLists* (page 293) GHC uses the `fromList` (etc) methods from module `GHC.Exts`. You do not need to import `GHC.Exts` for this to happen.

However if you use *RebindableSyntax* (page 295), then GHC instead uses whatever is in scope with the names of `toList`, `fromList` and `fromListN`. That is, these functions are rebindable; c.f. *Rebindable syntax and the implicit Prelude import* (page 295).

6.2.9.3 Defaulting

Currently, the `IsList` class is not accompanied with defaulting rules. Although feasible, not much thought has gone into how to specify the meaning of the default declarations like:

```
default ([a])
```

6.2.9.4 Speculation about the future

The current implementation of the `OverloadedLists` extension can be improved by handling the lists that are only populated with literals in a special way. More specifically, the compiler could allocate such lists statically using a compact representation and allow `IsList` instances to take advantage of the compact representation. Equipped with this capability the `OverloadedLists` extension will be in a good position to subsume the `OverloadedStrings` extension (currently, as a special case, string literals benefit from statically allocated compact representation).

6.2.10 Rebindable syntax and the implicit Prelude import

`NoImplicitPrelude`

Implied by *RebindableSyntax* (page 295)

Since 6.8.1

Don't import `Prelude` by default.

GHC normally imports `Prelude.hi` files for you. If you'd rather it didn't, then give it a `-XNoImplicitPrelude` option. The idea is that you can then import a Prelude of your own.

`RebindableSyntax`

Implies *NoImplicitPrelude* (page 295)

Since 7.0.1

Enable rebinding of a variety of usually-built-in operations.

Suppose you are importing a Prelude of your own in order to define your own numeric class hierarchy. It completely defeats that purpose if the literal "1" means "`Prelude.fromInteger 1`", which is what the Haskell Report specifies. So the *RebindableSyntax* (page 295) extension causes the following pieces of built-in syntax to refer to *whatever is in scope*, not the Prelude versions:

- An integer literal `368` means "`fromInteger (368::Integer)`", rather than "`Prelude.fromInteger (368::Integer)`".
- Fractional literals are handled in just the same way, except that the translation is `fromRational (3.68::Rational)`.
- String literals are also handled the same way, except that the translation is `fromString ("368"::String)`.
- The equality test in an overloaded numeric pattern uses whatever `(==)` is in scope.
- The subtraction operation, and the greater-than-or-equal test, in `n+k` patterns use whatever `(-)` and `(>=)` are in scope.
- Negation (e.g. "`- (f x)`") means "`negate (f x)`", both in numeric patterns, and expressions.
- Conditionals (e.g. "`if e1 then e2 else e3`") means "`ifThenElse e1 e2 e3`". However `case` expressions are unaffected.
- "Do" notation is translated using whatever functions `(>>=)`, `(>>)`, and `fail`, are in scope (not the Prelude versions). List comprehensions, `mdo` (*The recursive do-notation* (page 279)), and parallel array comprehensions, are unaffected.
- Arrow notation (see *Arrow notation* (page 311)) uses whatever `arr`, `(>>>)`, `first`, `app`, `(|||)` and `loop` functions are in scope. But unlike the other constructs, the types of these functions must match the Prelude types very closely. Details are in flux; if you want to use this, ask!
- List notation, such as `[x,y]` or `[m..n]` can also be treated via rebindable syntax if you use *-XOverloadedLists*; see *Overloaded lists* (page 293).
- An overloaded label "`#foo`" means "`fromLabel @"foo"`", rather than "`GHC.OverloadedLabels.fromLabel @"foo"`" (see *Overloaded labels* (page 496)).

`RebindableSyntax` (page 295) implies *`NoImplicitPrelude`* (page 295).

In all cases (apart from arrow notation), the static semantics should be that of the desugared form, even if that is a little unexpected. For example, the static semantics of the literal `368` is exactly that of `fromInteger (368::Integer)`; it's fine for `fromInteger` to have any of the types:

```
fromInteger :: Integer -> Integer
fromInteger :: forall a. Foo a => Integer -> a
fromInteger :: Num a => a -> Integer
fromInteger :: Integer -> Bool -> Bool
```

Be warned: this is an experimental facility, with fewer checks than usual. Use `-dcore-lint` to typecheck the desugared program. If Core Lint is happy you should be all right.

6.2.10.1 Custom Prelude modules named Prelude

If you call your custom Prelude module `Prelude` and place it in a file called `Prelude.hs`, then your custom Prelude will be implicitly imported instead of the default Prelude.

Here is an example that compiles:

```
$ cat Prelude.hs
module Prelude where

a = ()

$ cat B.hs
module B where

foo = a

$ ghc Prelude.hs B.hs
[1 of 2] Compiling Prelude          ( Prelude.hs, Prelude.o )
[2 of 2] Compiling B                ( B.hs, B.o )
```

The new `Prelude` is implicitly imported in `B.hs`.

Here is an example that does not compile:

```
$ cat Prelude.hs
module Prelude where

foo = True

$ ghc Prelude.hs
[1 of 1] Compiling Prelude          ( Prelude.hs, Prelude.o )

Prelude.hs:3:7: error: Data constructor not in scope: True
```

The original `Prelude` module is shadowed by the custom Prelude in this case. To include the original Prelude in your custom Prelude, you can explicitly import it with the `-XPackageImports` option and `import "base" Prelude`.

6.2.10.2 Things unaffected by RebindableSyntax

RebindableSyntax (page 295) does not apply to any code generated from a `deriving` clause or declaration. To see why, consider the following code:

```
{-# LANGUAGE RebindableSyntax, OverloadedStrings #-}
newtype Text = Text String

fromString :: String -> Text
fromString = Text

data Foo = Foo deriving Show
```

This will generate code to the effect of:

```
instance Show Foo where
  showsPrec _ Foo = showString "Foo"
```

But because *RebindableSyntax* (page 295) and *OverloadedStrings* (page 495) are enabled, the "Foo" string literal would now be of type Text, not String, which showString doesn't accept! This causes the generated Show instance to fail to typecheck. It's hard to imagine any scenario where it would be desirable have *RebindableSyntax* (page 295) behavior within derived code, so GHC simply ignores *RebindableSyntax* (page 295) entirely when checking derived code.

6.2.11 Postfix operators

PostfixOperators

Since 7.10.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow the use of post-fix operators

The *PostfixOperators* (page 298) extension enables a small extension to the syntax of left operator sections, which allows you to define postfix operators. The extension is this: for any expression e and operator (!), the left section

```
(e !)
```

is equivalent (from the point of view of both type checking and execution) to the expression

```
((!) e)
```

The strict Haskell 98 interpretation is that the section is equivalent to

```
(\y -> (!) e y)
```

That is, the operator must be a function of two arguments. GHC allows it to take only one argument, and that in turn allows you to write the function postfix.

The extension does not extend to the left-hand side of function definitions; you must define such a function in prefix form.

6.2.12 Tuple sections

TupleSections

Since 6.12

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow the use of tuple section syntax

The *TupleSections* (page 298) extension enables partially applied tuple constructors. For example, the following program

```
(, True)
```

is considered to be an alternative notation for the more unwieldy alternative

```
\x -> (x, True)
```

You can omit any combination of arguments to the tuple, as in the following

```
(, "I", , , "Love", , 1337)
```

which translates to

```
\a b c d -> (a, "I", b, c, "Love", d, 1337)
```

If you have *unboxed tuples* (page **??**) enabled, tuple sections will also be available for them, like so

```
(# , True #)
```

Because there is no unboxed unit tuple, the following expression

```
(# #)
```

continues to stand for the unboxed singleton tuple data constructor.

6.2.13 Lambda-case

LambdaCase

Since 7.6.1

Status Included in *GHC2024* (page 270)

Allow the use of lambda-case syntax.

The *LambdaCase* (page 299) extension enables expressions of the form

```
\case { p1 -> e1; ...; pN -> eN }
```

which is equivalent to

```
\freshName -> case freshName of { p1 -> e1; ...; pN -> eN }
```

Since GHC 9.4.1, it also allows expressions with multiple scrutinees (see GHC proposal #302) of the form

```
\cases { p11 ... pM1 -> e1; ...; p1N ... pMN -> eN }
```

which is equivalent to a function defined as

```
f p11 ... pM1 = e1
...
f p1N ... pMN = eN
```

Note that both `\case` and `\cases` start a layout, so you can write

```
\case
  p1 -> e1
  ...
  pN -> eN
```

Additionally, since GHC 9.0.1, combining *LambdaCase* (page 299) with *Arrows* (page 311) allows `\case` (and since GHC 9.4.1 `\cases`) syntax to be used as a command in `proc` notation:

```
proc x -> (f -< x) `catchA` \case
  p1 -> cmd1
  ...
  pN -> cmdN
```

6.2.14 Empty case alternatives

EmptyCase

Since 7.8.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow empty case expressions.

The *EmptyCase* (page 300) extension enables case expressions, or lambda-case expressions, that have no alternatives, thus:

```
case e of { }   -- No alternatives
```

or

```
\case { }       -- -XLambdaCase is also required
```

Note that it is not allowed for `\cases`, since it would be unclear how many patterns are being matched.

This can be useful when you know that the expression being scrutinised has no non-bottom values. For example:

```
data Void
f :: Void -> Int
f x = case x of { }
```

With dependently-typed features it is more useful (see #2431). For example, consider these two candidate definitions of `absurd`:

```
data a :~: b where
  Refl :: a :~: a

absurd :: True :~: False -> a
absurd x = error "absurd"    -- (A)
absurd x = case x of {}      -- (B)
```

We much prefer (B). Why? Because GHC can figure out that `(True :~: False)` is an empty type. So (B) has no partiality and GHC is able to compile with *-Wincomplete-patterns* (page 94) and *-Werror* (page 87). On the other hand (A) looks dangerous, and GHC doesn't check to make sure that, in fact, the function can never get called.

6.2.15 Multi-way if-expressions

MultiWayIf

Since 7.6.1

Allow the use of multi-way-`if` syntax.

With *MultiWayIf* (page 301) extension GHC accepts conditional expressions with multiple branches:

```
if | guard1 -> expr1
   | ...
   | guardN -> exprN
```

which is roughly equivalent to

```
case () of
  _ | guard1 -> expr1
  ...
  _ | guardN -> exprN
```

Multi-way if expressions introduce a new layout context. So the example above is equivalent to:

```
if { | guard1 -> expr1
   ; | ...
   ; | guardN -> exprN
   }
```

The following behaves as expected:

```
if | guard1 -> if | guard2 -> expr2
                  | guard3 -> expr3
   | guard4 -> expr4
```

because layout translates it as

```
if { | guard1 -> if { | guard2 -> expr2
                    ; | guard3 -> expr3
                    }
   ; | guard4 -> expr4
   }
```

Layout with multi-way if works in the same way as other layout contexts, except that the semi-colons between guards in a multi-way if are optional. So it is not necessary to line up all the guards at the same column; this is consistent with the way guards work in function definitions and case expressions.

Note that multi-way if supports guards other than boolean conditions:

```
if | parseNumbers settings
   , Just (exponent, mantissa) <- decomposeNumber str
   , let (integralPart, fractionPart) = parse mantissa
   , integralPart >= 0 = ...
   | otherwise = ...
```

6.2.16 Local Fixity Declarations

A careful reading of the Haskell 98 Report reveals that fixity declarations (`infix`, `infixl`, and `infixr`) are permitted to appear inside local bindings such those introduced by `let` and `where`. However, the Haskell Report does not specify the semantics of such bindings very precisely.

In GHC, a fixity declaration may accompany a local binding:

```
let f = ...
    infixr 3 `f`
in
    ...
```

and the fixity declaration applies wherever the binding is in scope. For example, in a `let`, it applies in the right-hand sides of other `let`-bindings and the body of the `letC`. Or, in recursive `do` expressions (*The recursive do-notation* (page 279)), the local fixity declarations of a `let` statement scope over other statements in the group, just as the bound name does.

Moreover, a local fixity declaration *must* accompany a local binding of that name: it is not possible to revise the fixity of name bound elsewhere, as in

```
let infixr 9 $ in ...
```

Because local fixity declarations are technically Haskell 98, no extension is necessary to enable them.

6.2.17 More liberal syntax for function arguments

`BlockArguments`

> **Since** 8.6.1
>
> Allow `do` expressions, lambda expressions, etc. to be directly used as a function argument.

In Haskell 2010, certain kinds of expressions can be used without parentheses as an argument to an operator, but not as an argument to a function. They include `do`, lambda, `if`, `case`, and `let` expressions. Some GHC extensions also define language constructs of this type: `mdo` (*The recursive do-notation* (page 279)), `\case` (*Lambda-case* (page 299)), and `proc` (*Arrow notation* (page 311)).

The *`BlockArguments`* (page 302) extension allows these constructs to be directly used as a function argument. For example:

```
when (x > 0) do
  print x
  exitFailure
```

will be parsed as:

```
when (x > 0) (do
  print x
  exitFailure)
```

and

```
withForeignPtr fptr \ptr -> c_memcpy buf ptr size
```

will be parsed as:

```
withForeignPtr fptr (\ptr -> c_memcpy buf ptr size)
```

6.2.17.1 Changes to the grammar

The Haskell report defines the `lexp` nonterminal thus (* indicates a rule of interest)

```
lexp  →  \ apat1 … apatn -> exp          (lambda abstraction, n ≥ 1)  *
      |  let decls in exp                (let expression)             *
      |  if exp [;] then exp [;] else exp  (conditional)              *
      |  case exp of { alts }            (case expression)            *
      |  do { stmts }                    (do expression)              *
      |  fexp

fexp  →  [fexp] aexp                     (function application)

aexp  →  qvar                            (variable)
      |  gcon                            (general constructor)
      |  literal
      |  ( exp )                         (parenthesized expression)
      |  qcon { fbind1 … fbindn }        (labeled construction)
      |  aexp { fbind1 … fbindn }        (labelled update)
      |  …
```

The *BlockArguments* (page 302) extension moves these production rules under `aexp`

```
lexp  →  fexp

fexp  →  [fexp] aexp                     (function application)

aexp  →  qvar                            (variable)
      |  gcon                            (general constructor)
      |  literal
      |  ( exp )                         (parenthesized expression)
      |  qcon { fbind1 … fbindn }        (labeled construction)
      |  aexp { fbind1 … fbindn }        (labelled update)
      |  \ apat1 … apatn -> exp          (lambda abstraction, n ≥ 1)  *
      |  let decls in exp                (let expression)             *
      |  if exp [;] then exp [;] else exp  (conditional)              *
      |  case exp of { alts }            (case expression)            *
      |  do { stmts }                    (do expression)              *
      |  …
```

Now the `lexp` nonterminal is redundant and can be dropped from the grammar.

Note that this change relies on an existing meta-rule to resolve ambiguities:

> The grammar is ambiguous regarding the extent of lambda abstractions, let expressions, and conditionals. The ambiguity is resolved by the meta-rule that each of these constructs extends as far to the right as possible.

For example, `f \a -> a b` will be parsed as `f (\a -> a b)`, not as `f (\a -> a) b`.

6.2.18 Typed Holes

Typed holes are a feature of GHC that allows special placeholders written with a leading underscore (e.g., "_", "_foo", "_bar"), to be used as expressions. During compilation these holes will generate an error message that describes which type is expected at the hole's location, information about the origin of any free type variables, and a list of local bindings that might help fill the hole and bindings in scope that fit the type of the hole that might help fill the hole with actual code. Typed holes are always enabled in GHC.

The goal of typed holes is to help with writing Haskell code rather than to change the type system. Typed holes can be used to obtain extra information from the type checker, which might otherwise be hard to get. Normally, using GHCi, users can inspect the (inferred) type signatures of all top-level bindings. However, this method is less convenient with terms that are not defined on top-level or inside complex expressions. Holes allow the user to check the type of the term they are about to write.

For example, compiling the following module with GHC:

```
f :: a -> a
f x = _
```

will fail with the following error:

```
hole.hs:2:7:
    Found hole `_' with type: a
    Where: `a' is a rigid type variable bound by
               the type signature for f :: a -> a at hole.hs:1:6
    In the expression: _
    In an equation for `f': f x = _
    Relevant bindings include
      x :: a (bound at hole.hs:2:3)
      f :: a -> a (bound at hole.hs:2:1)
    Valid hole fits include x :: a (bound at hole.hs:2:3)
```

Here are some more details:

- A "Found hole" error usually terminates compilation, like any other type error. After all, you have omitted some code from your program. Nevertheless, you can run and test a piece of code containing holes, by using the *-fdefer-typed-holes* (page 414) flag. This flag defers errors produced by typed holes until runtime, and converts them into compile-time warnings. These warnings can in turn be suppressed entirely by *-Wno-typed-holes* (page 88).

  The same behaviour for "Variable out of scope" errors, it terminates compilation by default. You can defer such errors by using the *-fdefer-out-of-scope-variables* (page 415) flag. This flag defers errors produced by out of scope variables until runtime, and converts them into compile-time warnings. These warnings can in turn be suppressed entirely by *-Wno-deferred-out-of-scope-variables* (page 88).

  The result is that a hole or a variable will behave like undefined, but with the added benefits that it shows a warning at compile time, and will show the same message if it gets evaluated at runtime. This behaviour follows that of the *-fdefer-type-errors* (page 414) option, which implies *-fdefer-typed-holes* (page 414) and *-fdefer-out-of-scope-variables* (page 415). See *Deferring type errors to runtime* (page 414).

- All unbound identifiers are treated as typed holes, *whether or not they start with an underscore*. The only difference is in the error message:

```
cons z = z : True : _x : y
```

yields the errors

```
Foo.hs:3:21: error:
   Found hole: _x :: Bool
   Or perhaps '_x' is mis-spelled, or not in scope
   In the first argument of '(:)', namely '_x'
   In the second argument of '(:)', namely '_x : y'
   In the second argument of '(:)', namely 'True : _x : y'
   Relevant bindings include
     z :: Bool (bound at Foo.hs:3:6)
   cons :: Bool -> [Bool] (bound at Foo.hs:3:1)
   Valid hole fits include
     z :: Bool (bound at mpt.hs:2:6)
     otherwise :: Bool
       (imported from 'Prelude' at mpt.hs:1:8-10
       (and originally defined in 'GHC.Base'))
     False :: Bool
       (imported from 'Prelude' at mpt.hs:1:8-10
       (and originally defined in 'GHC.Types'))
     True :: Bool
       (imported from 'Prelude' at mpt.hs:1:8-10
       (and originally defined in 'GHC.Types'))
     maxBound :: forall a. Bounded a => a
       with maxBound @Bool
       (imported from 'Prelude' at mpt.hs:1:8-10
       (and originally defined in 'GHC.Enum'))
     minBound :: forall a. Bounded a => a
       with minBound @Bool
       (imported from 'Prelude' at mpt.hs:1:8-10
       (and originally defined in 'GHC.Enum'))

Foo.hs:3:26: error:
    Variable not in scope: y :: [Bool]
```

More information is given for explicit holes (i.e. ones that start with an underscore), than for out-of-scope variables, because the latter are often unintended typos, so the extra information is distracting. If you want the detailed information, use a leading underscore to make explicit your intent to use a hole.

- Unbound identifiers with the same name are never unified, even within the same function, but shown individually. For example:

```
cons = _x : _x
```

results in the following errors:

```
unbound.hs:1:8:
    Found hole '_x' with type: a
    Where: `a' is a rigid type variable bound by
               the inferred type of cons :: [a] at unbound.hs:1:1
    In the first argument of `(:)', namely `_x'
    In the expression: _x : _x
    In an equation for `cons': cons = _x : _x
    Relevant bindings include cons :: [a] (bound at unbound.hs:1:1)
```

(continues on next page)

(continued from previous page)

```
unbound.hs:1:13:
    Found hole: _x :: [a]
    Where: ‘a’ is a rigid type variable bound by
            the inferred type of cons :: [a]
            at unbound.hs:3:1-12
    Or perhaps ‘_x’ is mis-spelled, or not in scope
    In the second argument of ‘(:)’, namely ‘_x’
    In the expression: _x : _x
    In an equation for ‘cons’: cons = _x : _x
    Relevant bindings include cons :: [a] (bound at unbound.hs:3:1)
    Valid hole fits include
      cons :: forall a. [a]
        with cons @a
        (defined at mpt.hs:3:1)
      mempty :: forall a. Monoid a => a
        with mempty @[a]
        (imported from ‘Prelude’ at mpt.hs:1:8-10
        (and originally defined in ‘GHC.Base’))
```

Notice the two different types reported for the two different occurrences of _x.

- No language extension is required to use typed holes. The lexeme "_" was previously illegal in Haskell, but now has a more informative error message. The lexeme "_x" is a perfectly legal variable, and its behaviour is unchanged when it is in scope. For example

```
f _x = _x + 1
```

does not elicit any errors. Only a variable *that is not in scope* (whether or not it starts with an underscore) is treated as an error (which it always was), albeit now with a more informative error message.

- Unbound data constructors used in expressions behave exactly as above. However, unbound data constructors used in *patterns* cannot be deferred, and instead bring compilation to a halt. (In implementation terms, they are reported by the renamer rather than the type checker.)
- The list of valid hole fits is found by checking which bindings in scope would fit into the hole. As an example, compiling the following module with GHC:

```
import Data.List (inits)

g :: [String]
g = _ "hello, world"
```

yields the errors:

```
• Found hole: _ :: [Char] -> [String]
• In the expression: _
  In the expression: _ "hello, world"
  In an equation for ‘g’: g = _ "hello, world"
• Relevant bindings include g :: [String] (bound at mpt.hs:6:1)
  Valid hole fits include
    lines :: String -> [String]
      (imported from ‘Prelude’ at mpt.hs:3:8-9
       (and originally defined in ‘base-4.11.0.0:Data.OldList’))
    words :: String -> [String]
      (imported from ‘Prelude’ at mpt.hs:3:8-9
```

(continues on next page)

(continued from previous page)

```
      (and originally defined in ‘base-4.11.0.0:Data.OldList’))
    inits :: forall a. [a] -> [[a]]
      with inits @Char
      (imported from ‘Data.List’ at mpt.hs:4:19-23
       (and originally defined in ‘base-4.11.0.0:Data.OldList’))
    repeat :: forall a. a -> [a]
      with repeat @String
      (imported from ‘Prelude’ at mpt.hs:3:8-9
       (and originally defined in ‘GHC.List’))
    fail :: forall (m :: * -> *). Monad m => forall a. String -> m a
      with fail @[] @String
      (imported from ‘Prelude’ at mpt.hs:3:8-9
       (and originally defined in ‘GHC.Base’))
    return :: forall (m :: * -> *). Monad m => forall a. a -> m a
      with return @[] @String
      (imported from ‘Prelude’ at mpt.hs:3:8-9
       (and originally defined in ‘GHC.Base’))
    pure :: forall (f :: * -> *). Applicative f => forall a. a -> f a
      with pure @[] @String
      (imported from ‘Prelude’ at mpt.hs:3:8-9
       (and originally defined in ‘GHC.Base’))
    read :: forall a. Read a => String -> a
      with read @[String]
      (imported from ‘Prelude’ at mpt.hs:3:8-9
       (and originally defined in ‘Text.Read’))
    mempty :: forall a. Monoid a => a
      with mempty @([Char] -> [String])
      (imported from ‘Prelude’ at mpt.hs:3:8-9
       (and originally defined in ‘GHC.Base’))
```

There are a few flags for controlling the amount of context information shown for typed holes:

-fshow-hole-constraints

When reporting typed holes, also print constraints that are in scope. Example:

```
f :: Eq a => a -> Bool
f x = _
```

results in the following message:

```
show_constraints.hs:4:7: error:
    • Found hole: _ :: Bool
    • In the expression: _
      In an equation for ‘f’: f x = _
    • Relevant bindings include
        x :: a (bound at show_constraints.hs:4:3)
        f :: a -> Bool (bound at show_constraints.hs:4:1)
      Constraints include Eq a (from show_constraints.hs:3:1-22)
      Valid hole fits include
        otherwise :: Bool
        False :: Bool
        True :: Bool
        maxBound :: forall a. Bounded a => a
          with maxBound @Bool
        minBound :: forall a. Bounded a => a
          with minBound @Bool
```

6.2.18.1 Valid Hole Fits

GHC sometimes suggests valid hole fits for typed holes, which is configurable by a few flags.

-fno-show-valid-hole-fits

Default off

This flag can be toggled to turn off the display of valid hole fits entirely.

-fmax-valid-hole-fits=⟨n⟩

Default 6

The list of valid hole fits is limited by displaying up to 6 hole fits per hole. The number of hole fits shown can be set by this flag. Turning the limit off with *-fno-max-valid-hole-fits* (page 308) displays all found hole fits.

-fshow-type-of-hole-fits

Default on

By default, the hole fits show the type of the hole fit. This can be turned off by the reverse of this flag.

-fshow-type-app-of-hole-fits

Default on

By default, the hole fits show the type application needed to make this hole fit fit the type of the hole, e.g. for the hole `(_ :: Int -> [Int])`, `mempty` is a hole fit with `mempty @(Int -> [Int])`. This can be toggled off with the reverse of this flag.

-fshow-docs-of-hole-fits

Default off

It can sometime be the case that the name and type of a valid hole fit is not enough to realize what the fit stands for. This flag adds the documentation of the fit to the message, if the documentation is available (and the module from which the function comes was compiled with the `-haddock` flag).

-fshow-type-app-vars-of-hole-fits

Default on

By default, the hole fits show the type application needed to make this hole fit fit the type of the hole, e.g. for the hole `(_ :: Int -> [Int])`, `mempty :: Monoid a => a` is a hole fit with `mempty @(Int -> [Int])`. This flag toggles whether to show `a ~ (Int -> [Int])` instead of `mempty @(Int -> [Int])` in the where clause of the valid hole fit message.

-fshow-provenance-of-hole-fits

Default on

By default, each hole fit shows the provenance information of its hole fit, i.e. where it was bound or defined, and what module it was originally defined in if it was imported. This can be toggled off using the reverse of this flag.

-funclutter-valid-hole-fits

Default off

This flag can be toggled to decrease the verbosity of the valid hole fit suggestions by not showing the provenance nor type application of the suggestions.

Refinement Hole Fits

When the flag *-frefinement-level-hole-fits=⟨n⟩* (page 310) is set to an `n` larger than `0`, GHC will offer up a list of valid refinement hole fits, which are valid hole fits that need up to `n` levels of additional refinement to be complete, where each level represents an additional hole in the hole fit that requires filling in. As an example, consider the hole in

```
f :: [Integer] -> Integer
f = _
```

When the refinement level is not set, it will only offer valid hole fits suggestions:

```
Valid hole fits include
  f :: [Integer] -> Integer
  head :: forall a. [a] -> a
    with head @Integer
  last :: forall a. [a] -> a
    with last @Integer
  maximum :: forall (t :: * -> *).
             Foldable t =>
             forall a. Ord a => t a -> a
    with maximum @[] @Integer
  minimum :: forall (t :: * -> *).
             Foldable t =>
             forall a. Ord a => t a -> a
    with minimum @[] @Integer
  product :: forall (t :: * -> *).
             Foldable t =>
             forall a. Num a => t a -> a
    with product @[] @Integer
  sum :: forall (t :: * -> *).
         Foldable t =>
         forall a. Num a => t a -> a
    with sum @[] @Integer
```

However, with *-frefinement-level-hole-fits=⟨n⟩* (page 310) set to e.g. *1*, it will additionally offer up a list of refinement hole fits, in this case:

```
Valid refinement hole fits include
  foldl1 (_ :: Integer -> Integer -> Integer)
    with foldl1 @[] @Integer
    where foldl1 :: forall (t :: * -> *).
                    Foldable t =>
                    forall a. (a -> a -> a) -> t a -> a
  foldr1 (_ :: Integer -> Integer -> Integer)
    with foldr1 @[] @Integer
    where foldr1 :: forall (t :: * -> *).
                    Foldable t =>
                    forall a. (a -> a -> a) -> t a -> a
  const (_ :: Integer)
    with const @Integer @[Integer]
    where const :: forall a b. a -> b -> a
  ($) (_ :: [Integer] -> Integer)
```

(continues on next page)

(continued from previous page)

```
    with ($) @GHC.Types.LiftedRep @[Integer] @Integer
    where ($) :: forall a b. (a -> b) -> a -> b
  fail (_ :: String)
    with fail @((->) [Integer]) @Integer
    where fail :: forall (m :: * -> *).
                  Monad m =>
                  forall a. String -> m a
  return (_ :: Integer)
    with return @((->) [Integer]) @Integer
    where return :: forall (m :: * -> *). Monad m => forall a. a -> m a
  (Some refinement hole fits suppressed;
    use -fmax-refinement-hole-fits=N or -fno-max-refinement-hole-fits)
```

Which shows that the hole could be replaced with e.g. `foldl1 _`. While not fixing the hole, this can help users understand what options they have.

-frefinement-level-hole-fits=⟨n⟩

Default off

The list of valid refinement hole fits is generated by considering hole fits with a varying amount of additional holes. The amount of holes in a refinement can be set by this flag. If the flag is set to 0 or not set at all, no valid refinement hole fits will be suggested.

-fabstract-refinement-hole-fits

Default off

Valid list of valid refinement hole fits can often grow large when the refinement level is `>= 2`, with holes like `head _ _` or `fst _ _`, which are valid refinements, but which are unlikely to be relevant since one or more of the holes are still completely open, in that neither the type nor kind of those holes are constrained by the proposed identifier at all. By default, such holes are not reported. By turning this flag on, such holes are included in the list of valid refinement hole fits.

-fmax-refinement-hole-fits=⟨n⟩

Default 6

The list of valid refinement hole fits is limited by displaying up to 6 hole fits per hole. The number of hole fits shown can be set by this flag. Turning the limit off with *-fno-max-refinement-hole-fits* (page 310) displays all found hole fits.

-fshow-hole-matches-of-hole-fits

Default on

The types of the additional holes in refinement hole fits are displayed in the output, e.g. `foldl1 (_ :: a -> a -> a)` is a refinement for the hole `_ :: [a] -> a`. If this flag is toggled off, the output will display only `foldl1 _`, which can be used as a direct replacement for the hole, without requiring `-XScopedTypeVariables`.

Sorting Valid Hole Fits

There are currently two ways to sort valid hole fits. Sorting can be toggled with *-fsort-valid-hole-fits* (page 311)

-fno-sort-valid-hole-fits

Default off

By default the valid hole fits are sorted to show the most relevant hole fits at the top of the list of valid hole fits. This can be toggled off with this flag.

-fsort-by-size-hole-fits

Default on

Sorts by how big the types the quantified type variables in the type of the function would have to be in order to match the type of the hole.

-fsort-by-subsumption-hole-fits

Default off

An alternative sort. Sorts by checking which hole fits subsume other hole fits, such that if hole fit a could be used as hole fits for hole fit b, then b appears before a in the output. It is more precise than the default sort, but also a lot slower, since a subsumption check has to be run for each pair of valid hole fits.

6.2.19 Arrow notation

Arrows

Since 6.8.1

Enable arrow notation.

Arrows are a generalisation of monads introduced by John Hughes. For more details, see

- "Generalising Monads to Arrows", John Hughes, in Science of Computer Programming 37, pp. 67–111, May 2000. The paper that introduced arrows: a friendly introduction, motivated with programming examples.
- "A New Notation for Arrows", Ross Paterson, in ICFP, Sep 2001. Introduced the notation described here.
- "Arrows and Computation", Ross Paterson, in The Fun of Programming, Palgrave, 2003.
- "Programming with Arrows", John Hughes, in 5th International Summer School on Advanced Functional Programming, Lecture Notes in Computer Science vol. 3622, Springer, 2004. This paper includes another introduction to the notation, with practical examples.
- "Type and Translation Rules for Arrow Notation in GHC", Ross Paterson and Simon Peyton Jones, September 16, 2004. A terse enumeration of the formal rules used (extracted from comments in the source code).
- The arrows web page at `https://www.haskell.org/arrows/` <https://www.haskell.org/arrows/>`__.

With the *Arrows* (page 311) extension, GHC supports the arrow notation described in the second of these papers, translating it using combinators from the Control.Arrow module. What follows is a brief introduction to the notation; it won't make much sense unless you've read Hughes's paper.

The extension adds a new kind of expression for defining arrows:

```
exp10 ::= ...
       |  proc apat -> cmd
```

where `proc` is a new keyword. The variables of the pattern are bound in the body of the `proc`-expression, which is a new sort of thing called a command. The syntax of commands is as follows:

```
cmd   ::= exp10 -<  exp
       |  exp10 -<< exp
       |  cmd0
```

with ⟨cmd⟩^{0} up to ⟨cmd⟩^{9} defined using infix operators as for expressions, and

```
cmd10 ::= \ apat ... apat -> cmd
       |  let decls in cmd
       |  if exp then cmd else cmd
       |  case exp of { calts }
       |  do { cstmt ; ... cstmt ; cmd }
       |  fcmd

fcmd  ::= fcmd aexp
       |  ( cmd )
       |  (| aexp cmd ... cmd |)

cstmt ::= let decls
       |  pat <- cmd
       |  rec { cstmt ; ... cstmt [;] }
       |  cmd
```

where ⟨calts⟩ are like ⟨alts⟩ except that the bodies are commands instead of expressions.

Commands produce values, but (like monadic computations) may yield more than one value, or none, and may do other things as well. For the most part, familiarity with monadic notation is a good guide to using commands. However the values of expressions, even monadic ones, are determined by the values of the variables they contain; this is not necessarily the case for commands.

A simple example of the new notation is the expression

```
proc x -> f -< x+1
```

We call this a procedure or arrow abstraction. As with a lambda expression, the variable `x` is a new variable bound within the `proc`-expression. It refers to the input to the arrow. In the above example, `-<` is not an identifier but a new reserved symbol used for building commands from an expression of arrow type and an expression to be fed as input to that arrow. (The weird look will make more sense later.) It may be read as analogue of application for arrows. The above example is equivalent to the Haskell expression

```
arr (\ x -> x+1) >>> f
```

That would make no sense if the expression to the left of `-<` involves the bound variable `x`. More generally, the expression to the left of `-<` may not involve any local variable, i.e. a variable bound in the current arrow abstraction. For such a situation there is a variant `-<<`, as in

```
proc x -> f x -<< x+1
```

which is equivalent to

```
arr (\ x -> (f x, x+1)) >>> app
```

so in this case the arrow must belong to the ArrowApply class. Such an arrow is equivalent to a monad, so if you're using this form you may find a monadic formulation more convenient.

6.2.19.1 do-notation for commands

Another form of command is a form of do-notation. For example, you can write

```
proc x -> do
        y <- f -< x+1
        g -< 2*y
        let z = x+y
        t <- h -< x*z
        returnA -< t+z
```

You can read this much like ordinary do-notation, but with commands in place of monadic expressions. The first line sends the value of x+1 as an input to the arrow f, and matches its output against y. In the next line, the output is discarded. The arrow returnA is defined in the Control.Arrow module as arr id. The above example is treated as an abbreviation for

```
arr (\ x -> (x, x)) >>>
        first (arr (\ x -> x+1) >>> f) >>>
        arr (\ (y, x) -> (y, (x, y))) >>>
        first (arr (\ y -> 2*y) >>> g) >>>
        arr snd >>>
        arr (\ (x, y) -> let z = x+y in ((x, z), z)) >>>
        first (arr (\ (x, z) -> x*z) >>> h) >>>
        arr (\ (t, z) -> t+z) >>>
        returnA
```

Note that variables not used later in the composition are projected out. After simplification using rewrite rules (see *Rewrite rules* (page 589)) defined in the Control.Arrow module, this reduces to

```
arr (\ x -> (x+1, x)) >>>
        first f >>>
        arr (\ (y, x) -> (2*y, (x, y))) >>>
        first g >>>
        arr (\ (_, (x, y)) -> let z = x+y in (x*z, z)) >>>
        first h >>>
        arr (\ (t, z) -> t+z)
```

which is what you might have written by hand. With arrow notation, GHC keeps track of all those tuples of variables for you.

Note that although the above translation suggests that let-bound variables like z must be monomorphic, the actual translation produces Core, so polymorphic variables are allowed.

It's also possible to have mutually recursive bindings, using the new rec keyword, as in the following example:

```
counter :: ArrowCircuit a => a Bool Int
counter = proc reset -> do
        rec     output <- returnA -< if reset then 0 else next
                next <- delay 0 -< output+1
        returnA -< output
```

The translation of such forms uses the `loop` combinator, so the arrow concerned must belong to the `ArrowLoop` class.

6.2.19.2 Conditional commands

In the previous example, we used a conditional expression to construct the input for an arrow. Sometimes we want to conditionally execute different commands, as in

```
proc (x,y) ->
        if f x y
        then g -< x+1
        else h -< y+2
```

which is translated to

```
arr (\ (x,y) -> if f x y then Left x else Right y) >>>
        (arr (\x -> x+1) >>> g) ||| (arr (\y -> y+2) >>> h)
```

Since the translation uses `|||`, the arrow concerned must belong to the `ArrowChoice` class.

There are also `case` commands, like

```
case input of
    [] -> f -< ()
    [x] -> g -< x+1
    x1:x2:xs -> do
        y <- h -< (x1, x2)
        ys <- k -< xs
        returnA -< y:ys
```

The syntax is the same as for `case` expressions, except that the bodies of the alternatives are commands rather than expressions. The translation is similar to that of `if` commands.

6.2.19.3 Defining your own control structures

As we're seen, arrow notation provides constructs, modelled on those for expressions, for sequencing, value recursion and conditionals. But suitable combinators, which you can define in ordinary Haskell, may also be used to build new commands out of existing ones. The basic idea is that a command defines an arrow from environments to values. These environments assign values to the free local variables of the command. Thus combinators that produce arrows from arrows may also be used to build commands from commands. For example, the `ArrowPlus` class includes a combinator

```
ArrowPlus a => (<+>) :: a b c -> a b c -> a b c
```

so we can use it to build commands:

```
expr' = proc x -> do
                returnA -< x
        <+> do
                symbol Plus -< ()
                y <- term -< ()
                expr' -< x + y
        <+> do
                symbol Minus -< ()
                y <- term -< ()
                expr' -< x - y
```

(The do on the first line is needed to prevent the first <+> ... from being interpreted as part of the expression on the previous line.) This is equivalent to

```
expr' = (proc x -> returnA -< x)
        <+> (proc x -> do
                symbol Plus -< ()
                y <- term -< ()
                expr' -< x + y)
        <+> (proc x -> do
                symbol Minus -< ()
                y <- term -< ()
                expr' -< x - y)
```

We are actually using <+> here with the more specific type

```
ArrowPlus a => (<+>) :: a (e,()) c -> a (e,()) c -> a (e,()) c
```

It is essential that this operator be polymorphic in e (representing the environment input to the command and thence to its subcommands) and satisfy the corresponding naturality property

```
arr (first k) >>> (f <+> g) = (arr (first k) >>> f) <+> (arr (first k) >>> g)
```

at least for strict k. (This should be automatic if you're not using seq.) This ensures that environments seen by the subcommands are environments of the whole command, and also allows the translation to safely trim these environments. (The second component of the input pairs can contain unnamed input values, as described in the next section.) The operator must also not use any variable defined within the current arrow abstraction.

We could define our own operator

```
untilA :: ArrowChoice a => a (e,s) () -> a (e,s) Bool -> a (e,s) ()
untilA body cond = proc x -> do
        b <- cond -< x
        if b then returnA -< ()
        else do
                body -< x
                untilA body cond -< x
```

and use it in the same way. Of course this infix syntax only makes sense for binary operators; there is also a more general syntax involving special brackets:

```
proc x -> do
        y <- f -< x+1
        (|untilA (increment -< x+y) (within 0.5 -< x)|)
```

6.2.19.4 Primitive constructs

Some operators will need to pass additional inputs to their subcommands. For example, in an arrow type supporting exceptions, the operator that attaches an exception handler will wish to pass the exception that occurred to the handler. Such an operator might have a type

```
handleA :: ... => a (e,s) c -> a (e,(Ex,s)) c -> a (e,s) c
```

where Ex is the type of exceptions handled. You could then use this with arrow notation by writing a command

```
body `handleA` \ ex -> handler
```

so that if an exception is raised in the command body, the variable ex is bound to the value of the exception and the command handler, which typically refers to ex, is entered. Though the syntax here looks like a functional lambda, we are talking about commands, and something different is going on. The input to the arrow represented by a command consists of values for the free local variables in the command, plus a stack of anonymous values. In all the prior examples, we made no assumptions about this stack. In the second argument to handleA, the value of the exception has been added to the stack input to the handler. The command form of lambda merely gives this value a name.

More concretely, the input to a command consists of a pair of an environment and a stack. Each value on the stack is paired with the remainder of the stack, with an empty stack being (). So operators like handleA that pass extra inputs to their subcommands can be designed for use with the notation by placing the values on the stack paired with the environment in this way. More precisely, the type of each argument of the operator (and its result) should have the form

```
a (e, (t1, ... (tn, ())...)) t
```

where ⟨e⟩ is a polymorphic variable (representing the environment) and ⟨ti⟩ are the types of the values on the stack, with ⟨t1⟩ being the "top". The polymorphic variable ⟨e⟩ must not occur in ⟨a⟩, ⟨ti⟩ or ⟨t⟩. However the arrows involved need not be the same. Here are some more examples of suitable operators:

```
bracketA :: ... => a (e,s) b -> a (e,(b,s)) c -> a (e,(c,s)) d -> a (e,s) d
runReader :: ... => a (e,s) c -> a' (e,(State,s)) c
runState :: ... => a (e,s) c -> a' (e,(State,s)) (c,State)
```

We can supply the extra input required by commands built with the last two by applying them to ordinary expressions, as in

```
proc x -> do
        s <- ...
        (|runReader (do { ... })|) s
```

which adds s to the stack of inputs to the command built using runReader.

The command versions of lambda abstraction and application are analogous to the expression versions. In particular, the beta and eta rules describe equivalences of commands. These three features (operators, lambda abstraction and application) are the core of the notation; everything else can be built using them, though the results would be somewhat clumsy. For example, we could simulate do-notation by defining

```
bind :: Arrow a => a (e,s) b -> a (e,(b,s)) c -> a (e,s) c
u `bind` f = returnA &&& u >>> f

bind_ :: Arrow a => a (e,s) b -> a (e,s) c -> a (e,s) c
u `bind_` f = u `bind` (arr fst >>> f)
```

We could simulate `if` by defining

```
cond :: ArrowChoice a => a (e,s) b -> a (e,s) b -> a (e,(Bool,s)) b
cond f g = arr (\ (e,(b,s)) -> if b then Left (e,s) else Right (e,s)) >>> f ||| g
```

6.2.19.5 Differences with the paper

- Instead of a single form of arrow application (arrow tail) with two translations, the implementation provides two forms `-<` (first-order) and `-<<` (higher-order).
- User-defined operators are flagged with banana brackets instead of a new `form` keyword.
- In the paper and the previous implementation, values on the stack were paired to the right of the environment in a single argument, but now the environment and stack are separate arguments.

6.2.19.6 Portability

Although only GHC implements arrow notation directly, there is also a preprocessor (available from the arrows web page) that translates arrow notation into Haskell 98 for use with other Haskell systems. You would still want to check arrow programs with GHC; tracing type errors in the preprocessor output is not easy. Modules intended for both GHC and the preprocessor must observe some additional restrictions:

- The module must import Control.Arrow.
- The preprocessor cannot cope with other Haskell extensions. These would have to go in separate modules.
- Because the preprocessor targets Haskell (rather than Core), `let`-bound variables are monomorphic.

6.2.20 Lexical negation

`LexicalNegation`

Since 9.0.1

Detect if the minus sign stands for negation during lexical analysis by checking for the surrounding whitespace.

In Haskell 2010, the minus sign stands for negation when it has no left-hand side. Consider `x = - 5` and `y = 2 - 5`. In `x`, there's no expression between the `=` and `-`, so the minus stands for negation, whereas in `y`, there's `2` to the left of the minus, therefore it stands for subtraction.

This leads to certain syntactic anomalies:

- `(% x)` is an operator section for any operator `(%)` except for `(-)`. `(- x)` is negated `x` rather than the right operator section of subtraction. Consequently, it is impossible to write such a section, and users are advised to write `(subtract x)` instead.

- Negative numbers must be parenthesized when they appear in function argument position. `f (-5)` is correct, whereas `f -5` is parsed as `(-) f 5`.

The latter issue is partly mitigated by *NegativeLiterals* (page 491). When it is enabled, `-5` is parsed as negative 5 regardless of context, so `f -5` works as expected. However, it only applies to literals, so `f -x` or `f -(a*2)` are still parsed as subtraction.

With *LexicalNegation* (page 317), both anomalies are resolved:

- `(% x)` is an operator section for any operator `(%)`, no exceptions, as long as there's whitespace between `%` and `x`.
- In `f -x`, the `-x` is parsed as the negation of `x` for any syntactically atomic expression `x` (variable, literal, or parenthesized expression).
- The prefix `-` binds tighter than any infix operator. `-a % b` is parsed as `(-a) % b` regardless of the fixity of `%`.

This means that `(- x)` is the right operator section of subtraction, whereas `(-x)` is the negation of `x`. Note that these expressions will often have different types (`(- x)` might have type `Int -> Int` while `(-x)` will have type `Int`), and so users mistaking one for the other will likely get a compile error.

Under *LexicalNegation* (page 317), negated literals are desugared without `negate`. That is, `-123` stands for `fromInteger (-123)` rather than `negate (fromInteger 123)`. This makes *LexicalNegation* (page 317) a valid replacement for *NegativeLiterals* (page 491).

6.3 Import and export

6.3.1 Hiding things the imported module doesn't export

Technically in Haskell 2010 this is illegal:

```
module A( f ) where
  f = True

module B where
  import A hiding( g )  -- A does not export g
  g = f
```

The `import A hiding( g )` in module `B` is technically an error (Haskell Report, 5.3.1) because `A` does not export `g`. However GHC allows it, in the interests of supporting backward compatibility; for example, a newer version of `A` might export `g`, and you want `B` to work in either case.

The warning *-Wdodgy-imports* (page 92), which is off by default but included with *-W* (page 86), warns if you hide something that the imported module does not export.

6.3.2 Package-qualified imports

PackageImports

Since 6.10.1

Allow the use of package-qualified `import` syntax.

With the *PackageImports* (page 319) extension, GHC allows import declarations to be qualified by the package name that the module is intended to be imported from. For example:

```
import "network" Network.Socket
```

would import the module `Network.Socket` from the package `network` (any version). This may be used to disambiguate an import when the same module is available from multiple packages, or is present in both the current package being built and an external package.

The special package name `this` can be used to refer to the current package being built.

Note: You probably don't need to use this feature, it was added mainly so that we can build backwards-compatible versions of packages when APIs change. It can lead to fragile dependencies in the common case: modules occasionally move from one package to another, rendering any package-qualified imports broken. See also *Thinning and renaming modules* (page 223) for an alternative way of disambiguating between module names.

6.3.3 Safe imports

With the *Safe* (page 586), *Trustworthy* (page 586) and *Unsafe* (page 587) language flags, GHC extends the import declaration syntax to take an optional `safe` keyword after the `import` keyword. This feature is part of the Safe Haskell GHC extension. For example:

```
import safe qualified Network.Socket as NS
```

would import the module `Network.Socket` with compilation only succeeding if `Network.Socket` can be safely imported. For a description of when a import is considered safe see *Safe Haskell* (page 577).

6.3.4 Explicit namespaces in import/export

ExplicitNamespaces

Implied by *TypeOperators* (page 323), *TypeFamilies* (page 338)

Since 7.6.1

Status Included in *GHC2024* (page 270)

Enable use of explicit namespaces in module export lists, patterns, and expressions.

In an import or export list, such as

```
module M( f, (++) ) where ...
  import N( f, (++) )
  ...
```

the entities f and (++) are *values*. However, with type operators (*Type operators* (page 323)) it becomes possible to declare (++) as a *type constructor*. In that case, how would you export or import it?

The *ExplicitNamespaces* (page 319) extension allows you to prefix the name of a type constructor in an import or export list with "type" to disambiguate this case, thus:

```
module M( f, type (++) ) where ...
  import N( f, type (++) )
  ...
module N( f, type (++) ) where
  data family a ++ b = L a | R b
```

The extension *ExplicitNamespaces* (page 319) is implied by *TypeOperators* (page 323) and (for some reason) by *TypeFamilies* (page 338).

In addition, with *PatternSynonyms* (page 461) you can prefix the name of a data constructor in an import or export list with the keyword pattern, to allow the import or export of a data constructor without its parent type constructor (see *Import and export of pattern synonyms* (page 465)).

Furthermore, *ExplicitNamespaces* (page 319) permits the use of the type keyword in patterns and expressions:

```
f (type t) x = ...        -- in a pattern
r = f (type Integer) 10   -- in an expression
```

This is used in conjunction with *RequiredTypeArguments* (page 395).

When *ExplicitNamespaces* (page 319) is enabled, it is possible to use the type and data keywords to specify the namespace of the name used in a fixity signature or a WARNING/DEPRECATED pragma. This can be useful for disambiguating between names in different namespaces that may conflict with each other.

Here is an example of using namespace specifiers to set different fixities for type-level and term-level operators:

```
type f $ a = f a
f $ a = f a

infixl 9 type $ -- type-level $ is left-associative with priority 9
infixr 0 data $ -- term-level $ is right-associative with priority 0
```

Similarly, it can be used in pragmas to deprecate only one name in a namespace:

```
data Solo = MkSolo

pattern Solo = MkSolo
{-# DEPRECATED data Solo "Use `MkSolo` instead" #-}

type family Head xs where
  Head (x : _) = x

pattern Head x <- (head -> x)

{-# WARNING in "x-partial" data Head "this is a partial type synonym" #-}
```

It is considered bad practice to use a fixity signature, WARNING pragma, or DEPRECATED pragma for a type-level name without an explicit type namespace, and doing so will become an error in a future version of GHC.

6.3.5 Writing qualified in postpositive position

ImportQualifiedPost

> **Since** 8.10.1
>
> **Status** Included in *GHC2024* (page 270), *GHC2021* (page 271)

`ImportQualifiedPost` allows the syntax `import M qualified`, that is, to annotate a module as qualified by writing `qualified` after the module name.

To import a qualified module usually you must specify `qualified` in prepositive position : `import qualified M`. This often leads to a "hanging indent" (which is automatically inserted by some autoformatters and common in many code bases. For example:

```
import qualified A
import           B
import           C
```

The `ImportQualifiedPost` extension allows `qualified` to appear in postpositive position : `import M qualified`. With this extension enabled, one can write:

```
import A qualified
import B
import C
```

It is an error if `qualified` appears in both pre and postpositive positions.

The warning *-Wprepositive-qualified-module* (page 88) (off by default) reports on any occurrences of imports annotated `qualified` using prepositive syntax.

6.4 Types

6.4.1 Data types with no constructors

EmptyDataDecls

> **Since** 6.8.1
>
> **Status** Included in *GHC2024* (page 270), *GHC2021* (page 271) and *Haskell2010* (page 272)

Allow definition of empty `data` types.

With the *EmptyDataDecls* (page 321) extension, GHC lets you declare a data type with no constructors.

You only need to enable this extension if the language you're using is Haskell 98, in which a data type must have at least one constructor. Haskell 2010 relaxed this rule to allow data types with no constructors, and thus *EmptyDataDecls* (page 321) is enabled by default when the language is Haskell 2010.

For example:

```
data S      -- S :: Type
data T a    -- T :: Type -> Type
```

Syntactically, the declaration lacks the "= constrs" part. The type can be parameterised over types of any kind, but if the kind is not `Type` then an explicit kind annotation must be used (see *Explicitly-kinded quantification* (page 511)).

Such data types have only one value, namely bottom. Nevertheless, they can be useful when defining "phantom types".

In conjunction with the *`EmptyDataDeriving`* (page 435) extension, empty data declarations can also derive instances of standard type classes (see *Deriving instances for empty data types* (page 435)).

6.4.2 Data type contexts

`DatatypeContexts`

Since 7.0.1

Status Deprecated, Included in *`Haskell98`* (page 272), *`Haskell2010`* (page 272)

Allow contexts on `data` types.

Haskell allows datatypes to be given contexts, e.g.

```
data Eq a => Set a = NilSet | ConsSet a (Set a)
```

give constructors with types:

```
NilSet :: Set a
ConsSet :: Eq a => a -> Set a -> Set a
```

This is widely considered a misfeature, and is going to be removed from the language. In GHC, it is controlled by the deprecated extension `DatatypeContexts`.

6.4.3 Infix type constructors, classes, and type variables

GHC allows type constructors, classes, and type variables to be operators, and to be written infix, very much like expressions. More specifically:

- A type constructor or class can be any non-reserved operator. Symbols used in types are always like capitalized identifiers; they are never variables. Note that this is different from the lexical syntax of data constructors, which are required to begin with a `:`.
- Data type and type-synonym declarations can be written infix, parenthesised if you want further arguments. E.g.

```
data a :*: b = Foo a b
type a :+: b = Either a b
class a :=: b where ...

data (a :**: b) x = Baz a b x
type (a :++: b) y = Either (a,b) y
```

- Types, and class constraints, can be written infix. For example

```
x :: Int :*: Bool
f :: (a :=: b) => a -> b
```

- Back-quotes work as for expressions, both for type constructors and type variables; e.g. `Int `Either` Bool`, or `Int `a` Bool`. Similarly, parentheses work the same; e.g. `(:*:) Int Bool`.
- Fixities may be declared for type constructors, or classes, just as for data constructors. However, one cannot distinguish between the two in a fixity declaration; a fixity declaration sets the fixity for a data constructor and the corresponding type constructor. For example:

  ```
  infixl 7 T, :*:
  ```

  sets the fixity for both type constructor `T` and data constructor `T`, and similarly for `:*:`.
- The function arrow `->` is `infixr` with fixity -1.

6.4.4 Type operators

TypeOperators

Implies *ExplicitNamespaces* (page 319)

Since 6.8.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow the use and definition of types with operator names.

The language *TypeOperators* (page 323) allows you to use infix operators in types.

- Operator symbols are *constructors* rather than type *variables* (as they are in terms).
- Operator symbols in types can be written infix, both in definitions and uses. For example:

  ```
  data a + b = Plus a b
  type Foo = Int + Bool
  ```

- Alphanumeric type constructors can now be written infix, using backquote syntax:

  ```
  x :: Int `Either` Bool
  x = Left 5
  ```

- There is now some potential ambiguity in import and export lists; for example if you write `import M( (+) )` do you mean the *function* `(+)` or the *type constructor* `(+)`? The default is the former, but with *ExplicitNamespaces* (page 319) (which is implied by *TypeOperators* (page 323)) GHC allows you to specify the latter by preceding it with the keyword `type`, thus:

  ```
  import M( type (+) )
  ```

  See *Explicit namespaces in import/export* (page 319).
- The fixity of a type operator may be set using the usual fixity declarations but, as in *Infix type constructors, classes, and type variables* (page 322), the function and type constructor share a single fixity.
- There is now potential ambiguity in the traditional syntax for data constructor declarations. For example:

```
type a :+: b = Either a b
data X = Int :+: Bool :+: Char
```

This code wants to declare both a type-level `:+:` and a term-level `:+:` (which is, generally, allowed). But we cannot tell how to parenthesize the data constructor declaration in X: either way makes sense. We might imagine that a fixity declaration could help us, but it is awkward to apply the fixity declaration to the very definition of a new data constructor. Instead of declaring delicate rules around this issue, GHC simply rejects if the top level of a traditional-syntax data constructor declaration uses two operators without parenthesizing.

6.4.5 Liberalised type synonyms

LiberalTypeSynonyms

Implies *ExplicitForAll* (page 506)

Since 6.8.1

Relax many of the Haskell 98 rules on type synonym definitions.

Type synonyms are like macros at the type level, but Haskell 98 imposes many rules on individual synonym declarations. With the *LiberalTypeSynonyms* (page 324) extension, GHC does validity checking on types *only after expanding type synonyms*. That means that GHC can be very much more liberal about type synonyms than Haskell 98.

- You can apply a type synonym to a forall type:

```
type Foo a = a -> a -> Bool

f :: Foo (forall b. b->b)
```

  After expanding the synonym, `f` has the legal (in GHC) type:

```
f :: (forall b. b->b) -> (forall b. b->b) -> Bool
```

- You can apply a type synonym to a partially applied type synonym:

```
type Generic i o = forall x. i x -> o x
type Id x = x

foo :: Generic Id []
```

  After expanding the synonym, `foo` has the legal (in GHC) type:

```
foo :: forall x. x -> [x]
```

GHC does kind checking before expanding synonyms.

After expanding type synonyms, GHC does validity checking on types, looking for the following malformedness which isn't detected simply by kind checking:

- Type constructor applied to a type involving for-alls (if *ImpredicativeTypes* (page 407) is off)
- Partially-applied type synonym.

So, for example, this will be rejected:

```
type Pr = forall a. a

h :: [Pr]
h = ...
```

because GHC does not allow type constructors applied to for-all types.

6.4.6 Existentially quantified data constructors

ExistentialQuantification

Implies *ExplicitForAll* (page 506)

Since 6.8.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow existentially quantified type variables in types.

The idea of using existential quantification in data type declarations was suggested by Perry, and implemented in Hope+ (Nigel Perry, *The Implementation of Practical Functional Programming Languages*, PhD Thesis, University of London, 1991). It was later formalised by Laufer and Odersky (*Polymorphic type inference and abstract data types*, TOPLAS, 16(5), pp. 1411-1430, 1994). It's been in Lennart Augustsson's `hbc` Haskell compiler for several years, and proved very useful. Here's the idea. Consider the declaration:

```
data Foo = forall a. MkFoo a (a -> Bool)
         | Nil
```

The data type `Foo` has two constructors with types:

```
MkFoo :: forall a. a -> (a -> Bool) -> Foo
Nil   :: Foo
```

Notice that the type variable `a` in the type of `MkFoo` does not appear in the data type itself, which is plain `Foo`. For example, the following expression is fine:

```
[MkFoo 3 even, MkFoo 'c' isUpper] :: [Foo]
```

Here, `(MkFoo 3 even)` packages an integer with a function `even` that maps an integer to `Bool`; and `MkFoo 'c' isUpper` packages a character with a compatible function. These two things are each of type `Foo` and can be put in a list.

What can we do with a value of type `Foo`? In particular, what happens when we pattern-match on `MkFoo`?

```
f (MkFoo val fn) = ???
```

Since all we know about `val` and `fn` is that they are compatible, the only (useful) thing we can do with them is to apply `fn` to `val` to get a boolean. For example:

```
f :: Foo -> Bool
f (MkFoo val fn) = fn val
```

What this allows us to do is to package heterogeneous values together with a bunch of functions that manipulate them, and then treat that collection of packages in a uniform manner. You can express quite a bit of object-oriented-like programming this way.

6.4.6.1 Why existential?

What has this to do with *existential* quantification? Simply that MkFoo has the (nearly) isomorphic type

```
MkFoo :: (exists a . (a, a -> Bool)) -> Foo
```

But Haskell programmers can safely think of the ordinary *universally* quantified type given above, thereby avoiding adding a new existential quantification construct.

6.4.6.2 Existentials and type classes

An easy extension is to allow arbitrary contexts before the constructor. For example:

```
data Baz = forall a. Eq a => Baz1 a a
         | forall b. Show b => Baz2 b (b -> b)
```

The two constructors have the types you'd expect:

```
Baz1 :: forall a. Eq a => a -> a -> Baz
Baz2 :: forall b. Show b => b -> (b -> b) -> Baz
```

But when pattern matching on Baz1 the matched values can be compared for equality, and when pattern matching on Baz2 the first matched value can be converted to a string (as well as applying the function to it). So this program is legal:

```
f :: Baz -> String
f (Baz1 p q) | p == q    = "Yes"
             | otherwise = "No"
f (Baz2 v fn)            = show (fn v)
```

Operationally, in a dictionary-passing implementation, the constructors Baz1 and Baz2 must store the dictionaries for Eq and Show respectively, and extract it on pattern matching.

6.4.6.3 Record Constructors

GHC allows existentials to be used with records syntax as well. For example:

```
data Counter a = forall self. NewCounter
    { _this    :: self
    , _inc     :: self -> self
    , _display :: self -> IO ()
    , tag      :: a
    }
```

Here tag is a public field, with a well-typed selector function tag :: Counter a -> a. See *Field selectors and TypeApplications* (page 421) for a full description of how the types of top-level field selectors are determined.

The self type is hidden from the outside; any attempt to apply _this, _inc or _display as functions will raise a compile-time error. In other words, *GHC defines a record selector function only for fields whose type does not mention the existentially-quantified variables*. (This example used an underscore in the fields for which record selectors will not be defined, but that is only programming style; GHC ignores them.)

To make use of these hidden fields, we need to create some helper functions:

```
inc :: Counter a -> Counter a
inc (NewCounter x i d t) = NewCounter
    { _this = i x, _inc = i, _display = d, tag = t }

display :: Counter a -> IO ()
display NewCounter{ _this = x, _display = d } = d x
```

Now we can define counters with different underlying implementations:

```
counterA :: Counter String
counterA = NewCounter
    { _this = 0, _inc = (1+), _display = print, tag = "A" }

counterB :: Counter String
counterB = NewCounter
    { _this = "", _inc = ('#':), _display = putStrLn, tag = "B" }

main = do
    display (inc counterA)         -- prints "1"
    display (inc (inc counterB))   -- prints "##"
```

Record update syntax is supported for existentials (and GADTs):

```
setTag :: Counter a -> a -> Counter a
setTag obj t = obj{ tag = t }
```

The rule for record update is this:

> the types of the updated fields may mention only the universally-quantified type variables of the data constructor. For GADTs, the field may mention only types that appear as a simple type-variable argument in the constructor's result type.

For example:

```
data T a b where { T1 { f1::a, f2::b, f3::(b,c) } :: T a b } -- c is existential
upd1 t x = t { f1=x }   -- OK:   upd1 :: T a b -> a' -> T a' b
upd2 t x = t { f3=x }   -- BAD   (f3's type mentions c, which is
                        --        existentially quantified)

data G a b where { G1 { g1::a, g2::c } :: G a [c] }
upd3 g x = g { g1=x }   -- OK:   upd3 :: G a b -> c -> G c b
upd4 g x = g { g2=x }   -- BAD (g2's type mentions c, which is not a simple
                        --      type-variable argument in G1's result type)
```

6.4.6.4 Restrictions

There are several restrictions on the ways in which existentially-quantified constructors can be used.

- When pattern matching, each pattern match introduces a new, distinct, type for each existential type variable. These types cannot be unified with any other type, nor can they escape from the scope of the pattern match. For example, these fragments are incorrect:

```
f1 (MkFoo a f) = a
```

Here, the type bound by `MkFoo` "escapes", because `a` is the result of `f1`. One way to see why this is wrong is to ask what type `f1` has:

```
f1 :: Foo -> a             -- Weird!
```

What is this "`a`" in the result type? Clearly we don't mean this:

```
f1 :: forall a. Foo -> a   -- Wrong!
```

The original program is just plain wrong. Here's another sort of error

```
f2 (Baz1 a b) (Baz1 p q) = a==q
```

It's ok to say `a==b` or `p==q`, but `a==q` is wrong because it equates the two distinct types arising from the two `Baz1` constructors.

- You can't pattern-match on an existentially quantified constructor in a `let` or `where` group of bindings. So this is illegal:

```
f3 x = a==b where { Baz1 a b = x }
```

Instead, use a `case` expression:

```
f3 x = case x of Baz1 a b -> a==b
```

In general, you can only pattern-match on an existentially-quantified constructor in a `case` expression or in the patterns of a function definition. The reason for this restriction is really an implementation one. Type-checking binding groups is already a nightmare without existentials complicating the picture. Also an existential pattern binding at the top level of a module doesn't make sense, because it's not clear how to prevent the existentially-quantified type "escaping". So for now, there's a simple-to-state restriction. We'll see how annoying it is.

- You can't use existential quantification for `newtype` declarations. So this is illegal:

```
newtype T = forall a. Ord a => MkT a
```

Reason: a value of type `T` must be represented as a pair of a dictionary for `Ord t` and a value of type `t`. That contradicts the idea that `newtype` should have no concrete representation. You can get just the same efficiency and effect by using `data` instead of `newtype`. If there is no overloading involved, then there is more of a case for allowing an existentially-quantified `newtype`, because the `data` version does carry an implementation cost, but single-field existentially quantified constructors aren't much use. So the simple restriction (no existential stuff on `newtype`) stands, unless there are convincing reasons to change it.

- You can't use `deriving` to define instances of a data type with existentially quantified data constructors. Reason: in most cases it would not make sense. For example:

```
data T = forall a. MkT [a] deriving( Eq )
```

To derive `Eq` in the standard way we would need to have equality between the single component of two `MkT` constructors:

```
instance Eq T where
  (MkT a) == (MkT b) = ???
```

But a and b have distinct types, and so can't be compared. It's just about possible to imagine examples in which the derived instance would make sense, but it seems altogether simpler simply to prohibit such declarations. Define your own instances!

6.4.7 Declaring data types with explicit constructor signatures

GADTSyntax

Implied by *GADTs* (page 335)

Since 7.2.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow the use of GADT syntax in data type definitions (but not GADTs themselves; for this see *GADTs* (page 335))

When the GADTSyntax extension is enabled, GHC allows you to declare an algebraic data type by giving the type signatures of constructors explicitly. For example:

```
data Maybe a where
    Nothing :: Maybe a
    Just    :: a -> Maybe a

newtype Down a where
  Down :: a -> Down a
```

The form is called a "GADT-style declaration" because Generalised Algebraic Data Types, described in *Generalised Algebraic Data Types (GADTs)* (page 335), can only be declared using this form.

Notice that GADT-style syntax generalises existential types (*Existentially quantified data constructors* (page 325)). For example, these two declarations are equivalent:

```
data Foo = forall a. MkFoo a (a -> Bool)
data Foo' where { MKFoo :: a -> (a->Bool) -> Foo' }
```

Any datatype (or newtype) that can be declared in standard Haskell 98 syntax, can also be declared using GADT-style syntax. The choice is largely stylistic, but GADT-style declarations differ in one important respect: they treat class constraints on the data constructors differently. Specifically, if the constructor is given a type-class context, that context is made available by pattern matching. For example:

```
data Set a where
  MkSet :: Eq a => [a] -> Set a

makeSet :: Eq a => [a] -> Set a
makeSet xs = MkSet (nub xs)

insert :: a -> Set a -> Set a
insert a (MkSet as) | a `elem` as = MkSet as
                    | otherwise   = MkSet (a:as)
```

A use of MkSet as a constructor (e.g. in the definition of makeSet) gives rise to a (Eq a) constraint, as you would expect. The new feature is that pattern-matching on MkSet (as in the definition of insert) makes *available* an (Eq a) context. In implementation terms, the MkSet constructor has a hidden field that stores the (Eq a) dictionary that is passed to MkSet; so when pattern-matching that dictionary becomes available for the right-hand side of the match.

In the example, the equality dictionary is used to satisfy the equality constraint generated by the call to `elem`, so that the type of `insert` itself has no `Eq` constraint.

For example, one possible application is to reify dictionaries:

```
data NumInst a where
  MkNumInst :: Num a => NumInst a

intInst :: NumInst Int
intInst = MkNumInst

plus :: NumInst a -> a -> a -> a
plus MkNumInst p q = p + q
```

Here, a value of type `NumInst a` is equivalent to an explicit `(Num a)` dictionary.

All this applies to constructors declared using the syntax of *Existentials and type classes* (page 326). For example, the `NumInst` data type above could equivalently be declared like this:

```
data NumInst a
   = Num a => MkNumInst (NumInst a)
```

Notice that, unlike the situation when declaring an existential, there is no `forall`, because the `Num` constrains the data type's universally quantified type variable `a`. A constructor may have both universal and existential type variables: for example, the following two declarations are equivalent:

```
data T1 a
 = forall b. (Num a, Eq b) => MkT1 a b
data T2 a where
 MkT2 :: (Num a, Eq b) => a -> b -> T2 a
```

All this behaviour contrasts with Haskell 98's peculiar treatment of contexts on a data type declaration (Section 4.2.1 of the Haskell 98 Report). In Haskell 98 the definition

```
data Eq a => Set' a = MkSet' [a]
```

gives `MkSet'` the same type as `MkSet` above. But instead of *making available* an `(Eq a)` constraint, pattern-matching on `MkSet'` *requires* an `(Eq a)` constraint! GHC faithfully implements this behaviour, odd though it is. But for GADT-style declarations, GHC's behaviour is much more useful, as well as much more intuitive.

Note that the restrictions of *Restrictions* (page 327) are still in place; for example, a newtype declared with `GADTSyntax` cannot use existential quantification.

6.4.7.1 Formal syntax for GADTs

To make more precise what is and what is not permitted inside of a GADT-style constructor, we provide a BNF-style grammar for GADT below. Note that this grammar is subject to change in the future.

```
gadt_con ::= conids '::' opt_forall opt_ctxt gadt_body

conids ::= conid
        |  conid ',' conids

```

(continues on next page)

(continued from previous page)

```
opt_forall ::= <empty>
            |  'forall' tv_bndrs '.'

tv_bndrs ::= <empty>
          |  tv_bndr tv_bndrs

tv_bndr ::= tyvar
         |  '(' tyvar '::' ctype ')'

opt_ctxt ::= <empty>
          |  btype '=>'
          |  '(' ctxt ')' '=>'

ctxt ::= ctype
      |  ctype ',' ctxt

gadt_body ::= prefix_gadt_body
           |  record_gadt_body

prefix_gadt_body ::= '(' prefix_gadt_body ')'
                  |  return_type
                  |  opt_unpack btype '->' prefix_gadt_body

record_gadt_body ::= '{' fieldtypes '}' '->' return_type

fieldtypes ::= <empty>
            |  fieldnames '::' opt_unpack ctype
            |  fieldnames '::' opt_unpack ctype ',' fieldtypes

fieldnames ::= fieldname
            |  fieldname ',' fieldnames

opt_unpack ::= opt_bang
            :  {-# UNPACK #-} opt_bang
            |  {-# NOUNPACK #-} opt_bang

opt_bang ::= <empty>
          |  '!'
          |  '~'
```

Where:

- `btype` is a type that is not allowed to have an outermost `forall`/`=>` unless it is surrounded by parentheses. For example, `forall a. a` and `Eq a => a` are not legal `btype`s, but `(forall a. a)` and `(Eq a => a)` are legal.
- `ctype` is a `btype` that has no restrictions on an outermost `forall`/`=>`, so `forall a. a` and `Eq a => a` are legal `ctype`s.
- `return_type` is a type that is not allowed to have `forall`s, `=>`s, or `->`s.

This is a simplified grammar that does not fully delve into all of the implementation details of GHC's parser (such as the placement of Haddock comments), but it is sufficient to attain an understanding of what is syntactically allowed. Some further various observations about this grammar:

- GADT constructor types are currently not permitted to have nested `forall`s or `=>`s. (e.g., something like `MkT :: Int -> forall a. a -> T` would be rejected.) As a result, `gadt_sig` puts all of its quantification and constraints up front with `opt_forall` and `opt_context`. Note that higher-rank `forall`s and `=>`s are only permitted if they do not appear directly to the right of a function arrow in a *prefix_gadt_body*. (e.g., something like `MkS :: Int -> (forall a. a) -> S` is allowed, since parentheses separate the `forall` from the `->`.)

- Furthermore, GADT constructors do not permit outermost parentheses that surround the `opt_forall` or `opt_ctxt`, if at least one of them are used. For example, `MkU :: (forall a. a -> U)` would be rejected, since it would treat the `forall` as being nested.

  Note that it is acceptable to use parentheses in a `prefix_gadt_body`. For instance, `MkV1 :: forall a. (a) -> (V1)` is acceptable, as is `MkV2 :: forall a. (a -> V2)`.

- The function arrows in a `prefix_gadt_body`, as well as the function arrow in a `record_gadt_body`, are required to be used infix. For example, `MkA :: (->) Int A` would be rejected.

- GHC uses the function arrows in a `prefix_gadt_body` and `prefix_gadt_body` to syntactically demarcate the function and result types. Note that GHC does not attempt to be clever about looking through type synonyms here. If you attempt to do this, for instance:

```
type C = Int -> B

data B where
  MkB :: C
```

  Then GHC will interpret the return type of `MkB` to be `C`, and since GHC requires that the return type must be headed by `B`, this will be rejected. On the other hand, it is acceptable to use type synonyms within the argument and result types themselves, so the following is permitted:

```
type B1 = Int
type B2 = B

data B where
  MkB :: B1 -> B2
```

- GHC will accept any combination of `!`/`~` and `{-# UNPACK #-}`/`{-# NOUNPACK #-}`, although GHC will ignore some combinations. For example, GHC will produce a warning if you write `{-# UNPACK #-} ~Int` and proceed as if you had written `Int`.

6.4.7.2 GADT syntax odds and ends

The rest of this section gives further details about GADT-style data type declarations.

- The result type of each data constructor must begin with the type constructor being defined. If the result type of all constructors has the form `T a1 ... an`, where `a1 ... an` are distinct type variables, then the data type is *ordinary*; otherwise is a *generalised* data type (*Generalised Algebraic Data Types (GADTs)* (page 335)).

- As with other type signatures, you can give a single signature for several data constructors. In this example we give a single signature for `T1` and `T2`:

```
data T a where
  T1,T2 :: a -> T a
  T3 :: T a
```

- The type signature of each constructor is independent, and is implicitly universally quantified as usual. In particular, the type variable(s) in the "`data T a where`" header have no scope, and different constructors may have different universally-quantified type variables:

```
data T a where         -- The 'a' has no scope
  T1,T2 :: b -> T b    -- Means forall b. b -> T b
  T3 :: T a            -- Means forall a. T a
```

- A constructor signature may mention type class constraints, which can differ for different constructors. For example, this is fine:

```
data T a where
  T1 :: Eq b => b -> b -> T b
  T2 :: (Show c, Ix c) => c -> [c] -> T c
```

  When pattern matching, these constraints are made available to discharge constraints in the body of the match. For example:

```
f :: T a -> String
f (T1 x y) | x==y      = "yes"
           | otherwise = "no"
f (T2 a b)             = show a
```

  Note that `f` is not overloaded; the `Eq` constraint arising from the use of `==` is discharged by the pattern match on `T1` and similarly the `Show` constraint arising from the use of `show`.

- Unlike a Haskell-98-style data type declaration, the type variable(s) in the "`data Set a where`" header have no scope. Indeed, one can write a kind signature instead:

```
data Set :: Type -> Type where ...
```

  or even a mixture of the two:

```
data Bar a :: (Type -> Type) -> Type where ...
```

  The type variables (if given) may be explicitly kinded, so we could also write the header for `Foo` like this:

```
data Bar a (b :: Type -> Type) where ...
```

- You can use strictness annotations, in the obvious places in the constructor type:

```
data Term a where
    Lit    :: !Int -> Term Int
    If     :: Term Bool -> !(Term a) -> !(Term a) -> Term a
    Pair   :: Term a -> Term b -> Term (a,b)
```

- You can use a `deriving` clause on a GADT-style data type declaration. For example, these two declarations are equivalent

```
data Maybe1 a where {
    Nothing1 :: Maybe1 a ;
    Just1    :: a -> Maybe1 a
  } deriving( Eq, Ord )
```

(continues on next page)

(continued from previous page)

```
data Maybe2 a = Nothing2 | Just2 a
     deriving( Eq, Ord )
```

- The type signature may have quantified type variables that do not appear in the result type:

```
data Foo where
   MkFoo :: a -> (a->Bool) -> Foo
   Nil   :: Foo
```

  Here the type variable `a` does not appear in the result type of either constructor. Although it is universally quantified in the type of the constructor, such a type variable is often called "existential". Indeed, the above declaration declares precisely the same type as the `data Foo` in *Existentially quantified data constructors* (page 325).

  The type may contain a class context too, of course:

```
data Showable where
  MkShowable :: Show a => a -> Showable
```

- You can use record syntax on a GADT-style data type declaration:

```
data Person where
    Adult :: { name :: String, children :: [Person] } -> Person
    Child :: Show a => { name :: !String, funny :: a } -> Person
```

  As usual, for every constructor that has a field `f`, the type of field `f` must be the same (modulo alpha conversion). The `Child` constructor above shows that the signature may have a context, existentially-quantified variables, and strictness annotations, just as in the non-record case. (NB: the "type" that follows the double-colon is not really a type, because of the record syntax and strictness annotations. A "type" of this form can appear only in a constructor signature.)

- Record updates are allowed with GADT-style declarations, only fields that have the following property: the type of the field mentions no existential type variables.

- As in the case of existentials declared using the Haskell-98-like record syntax (*Record Constructors* (page 326)), record-selector functions are generated only for those fields that have well-typed selectors. Here is the example of that section, in GADT-style syntax:

```
data Counter a where
    NewCounter :: { _this    :: self
                  , _inc     :: self -> self
                  , _display :: self -> IO ()
                  , tag      :: a
                  } -> Counter a
```

  As before, only one selector function is generated here, that for `tag`. Nevertheless, you can still use all the field names in pattern matching and record construction.

- In a GADT-style data type declaration there is no obvious way to specify that a data constructor should be infix, which makes a difference if you derive `Show` for the type. (Data constructors declared infix are displayed infix by the derived `show`.) So GHC implements the following design: a data constructor declared in a GADT-style data type declaration is displayed infix by `Show` iff (a) it is an operator symbol, (b) it has two arguments, (c) it has a programmer-supplied fixity declaration. For example

```
infix 6 (:--:)
data T a where
  (:--:) :: Int -> Bool -> T Int
```

6.4.8 Generalised Algebraic Data Types (GADTs)

GADTs

Implies *MonoLocalBinds* (page 526), *GADTSyntax* (page 329)

Since 6.8.1

Status Included in *GHC2024* (page 270)

Allow use of Generalised Algebraic Data Types (GADTs).

Generalised Algebraic Data Types generalise ordinary algebraic data types by allowing constructors to have richer return types. Here is an example:

```
data Term a where
    Lit    :: Int -> Term Int
    Succ   :: Term Int -> Term Int
    IsZero :: Term Int -> Term Bool
    If     :: Term Bool -> Term a -> Term a -> Term a
    Pair   :: Term a -> Term b -> Term (a,b)
```

Notice that the return type of the constructors is not always `Term a`, as is the case with ordinary data types. This generality allows us to write a well-typed `eval` function for these `Terms`:

```
eval :: Term a -> a
eval (Lit i)      = i
eval (Succ t)     = 1 + eval t
eval (IsZero t)   = eval t == 0
eval (If b e1 e2) = if eval b then eval e1 else eval e2
eval (Pair e1 e2) = (eval e1, eval e2)
```

The key point about GADTs is that *pattern matching causes type refinement*. For example, in the right hand side of the equation

```
eval :: Term a -> a
eval (Lit i) =  ...
```

the type `a` is refined to `Int`. That's the whole point! A precise specification of the type rules is beyond what this user manual aspires to, but the design closely follows that described in the paper Simple unification-based type inference for GADTs (ICFP 2006). The general principle is this: *type refinement is only carried out based on user-supplied type annotations*. So if no type signature is supplied for `eval`, no type refinement happens, and lots of obscure error messages will occur. However, the refinement is quite general. For example, if we had:

```
eval :: Term a -> a -> a
eval (Lit i) j =  i+j
```

the pattern match causes the type `a` to be refined to `Int` (because of the type of the constructor `Lit`), and that refinement also applies to the type of `j`, and the result type of the `case` expression. Hence the addition `i+j` is legal.

These and many other examples are given in papers by Hongwei Xi, and Tim Sheard. There is a longer introduction on the wiki, and Ralf Hinze's Fun with phantom types also has a number of examples. Note that papers may use different notation to that implemented in GHC.

The rest of this section outlines the extensions to GHC that support GADTs. The extension is enabled with *GADTs* (page 335). The *GADTs* (page 335) extension also sets *GADTSyntax* (page 329) and *MonoLocalBinds* (page 526).

- A GADT can only be declared using GADT-style syntax (*Declaring data types with explicit constructor signatures* (page 329)); the old Haskell 98 syntax for data declarations always declares an ordinary data type. The result type of each constructor must begin with the type constructor being defined, but for a GADT the arguments to the type constructor can be arbitrary monotypes. For example, in the `Term` data type above, the type of each constructor must end with `Term ty`, but the `ty` need not be a type variable (e.g. the `Lit` constructor).
- GADT constructors can include contexts and existential variables, generalising existential quantification (*Existentially quantified data constructors* (page 325)). For example:

```
data SomeShow where
    SomeShow :: Show a => a -> SomeShow
      -- `a` is existential, as it does not appear in the return type

data G a where
    MkG :: (a ~ Int) => a -> a -> G a
 -- essentially the same as:
 -- MkG :: Int -> Int -> G Int
```

- It is permitted to declare an ordinary algebraic data type using GADT-style syntax. What makes a GADT into a GADT is not the syntax, but rather the presence of data constructors whose result type is not just `T a b`, or which include contexts.
- A newtype may use GADT-style syntax, but it must declare an ordinary data type, not a GADT. That is, the constructor must not bind existential variables (as per *Existentially quantified data constructors* (page 325)) nor include a context.
- You cannot use a `deriving` clause for a GADT; only for an ordinary data type (possibly using GADT-style syntax). However, you can still use a *Stand-alone deriving declarations* (page 436) declaration.
- As mentioned in *Declaring data types with explicit constructor signatures* (page 329), record syntax is supported. For example:

```
data Term a where
    Lit    :: { val  :: Int }      -> Term Int
    Succ   :: { num  :: Term Int } -> Term Int
    Pred   :: { num  :: Term Int } -> Term Int
    IsZero :: { arg  :: Term Int } -> Term Bool
    Pair   :: { arg1 :: Term a
              , arg2 :: Term b
              }                    -> Term (a,b)
    If     :: { cnd  :: Term Bool
              , tru  :: Term a
              , fls  :: Term a
              }                    -> Term a
```

  However, for GADTs there is the following additional constraint: every constructor that has a field `f` must have the same result type (modulo alpha conversion) Hence, in the above example, we cannot merge the `num` and `arg` fields above into a single name. Although their field types are both `Term Int`, their selector functions actually have different types:

```
num :: Term Int -> Term Int
arg :: Term Bool -> Term Int
```

See *Field selectors and TypeApplications* (page 421) for a full description of how the types of top-level field selectors are determined.

- When pattern-matching against data constructors drawn from a GADT, for example in a `case` expression, the following rules apply:

  - The type of the scrutinee must be rigid.
  - The type of the entire `case` expression must be rigid.
  - The type of any free variable mentioned in any of the `case` alternatives must be rigid.

  A type is "rigid" if it is completely known to the compiler at its binding site. The easiest way to ensure that a variable has a rigid type is to give it a type signature. For more precise details see Simple unification-based type inference for GADTs. The criteria implemented by GHC are given in the Appendix.

- When GHC typechecks multiple patterns in a function clause, it typechecks each pattern in order from left to right. This has consequences for patterns that match on GADTs, such as in this example:

```
data U a where
  MkU :: U ()

v1 :: U a -> a -> a
v1 MkU () = ()

v2 :: a -> U a -> a
v2 () MkU = ()
```

  Although `v1` and `v2` may appear to be the same function but with differently ordered arguments, GHC will only typecheck `v1`. This is because in `v1`, GHC will first typecheck the `MkU` pattern, which causes `a` to be refined to `()`. This refinement is what allows the subsequent `()` pattern to typecheck at type `a`. In `v2`, however, GHC first tries to typecheck the `()` pattern, and because `a` has not been refined to `()` yet, GHC concludes that `()` is not of type `a`. `v2` can be made to typecheck by matching on `MkU` before `()`, like so:

```
v2 :: a -> U a -> a
v2 x MkU = case x of () -> ()
```

- Not only does GHC typecheck patterns from left to right, it also typechecks them from the outside in. This can be seen in this example:

```
data F x y where
  MkF :: y -> F (Maybe z) y

g :: F a a -> a
g (MkF Nothing) = Nothing
```

  In the function clause for `g`, GHC first checks `MkF`, the outermost pattern, followed by the inner `Nothing` pattern. This outside-in order can interact somewhat counterintuitively with *Pattern type signatures* (page 514). Consider the following variation of g:

```
g2 :: F a a -> a
g2 (MkF Nothing :: F (Maybe z) (Maybe z)) = Nothing @z
```

The g2 function attempts to use the pattern type signature F (Maybe z) (Maybe z) to bring the type variable z into scope so that it can be used on the right-hand side of the definition with *Visible type application* (page 386). However, GHC will reject the pattern type signature in g2:

```
• Couldn't match type ‘a’ with ‘Maybe z’
  Expected: F a a
    Actual: F (Maybe z) (Maybe z)
```

Again, this is because of the outside-in order GHC uses when typechecking patterns. GHC first tries to check the pattern type signature F (Maybe z) (Maybe z), but at that point, GHC has not refined a to be Maybe z, so GHC is unable to conclude that F a a is equal to F (Maybe z) (Maybe z). Here, the MkF pattern is considered to be inside of the pattern type signature, so GHC cannot use the type refinement from the MkF pattern when typechecking the pattern type signature.

There are two possible ways to repair g2. One way is to use a case expression to write a pattern signature *after* matching on MkF, like so:

```
g3 :: F a a -> a
g3 f@(MkF Nothing) =
  case f of
    (_ :: F (Maybe z) (Maybe z)) -> Nothing @z
```

Another way is to use *Type Abstractions in Patterns* (page 390) instead of a pattern type signature:

```
g4 :: F a a -> a
g4 (MkF @(Maybe z) Nothing) = Nothing @z
```

Here, the visible type argument @(Maybe z) indicates that the y in the type of MkF :: y -> F (Maybe z) y should be instantiated to Maybe z. In addition, @(Maybe z) also brings z into scope. Although g4 no longer uses a pattern type signature, it accomplishes the same end result, as the right-hand side Nothing @z will typecheck successfully.

6.4.9 Type families

TypeFamilies

Implies *MonoLocalBinds* (page 526), *KindSignatures* (page 511), *ExplicitNamespaces* (page 319)

Implied by *TypeFamilyDependencies* (page 355)

Since 6.8.1

Allow use and definition of indexed type and data families.

Indexed type families form an extension to facilitate type-level programming. Type families are a generalisation of associated data types [AssocDataTypes2005] and associated type synonyms [AssocTypeSyn2005] Type families themselves are described in Schrijvers 2008 [TypeFamilies2008]. Type families essentially provide type-indexed data types and named functions on types, which are useful for generic programming and highly parameterised library interfaces as well as interfaces with enhanced static information, much like dependent types. They might also be regarded as an alternative to functional dependencies, but provide a more functional style of type-level programming than the relational style of functional dependencies.

Indexed type families, or type families for short, are type constructors that represent sets of types. Set members are denoted by supplying the type family constructor with type parameters, which are called type indices. The difference between vanilla parametrised type constructors and family constructors is much like between parametrically polymorphic functions and (ad-hoc polymorphic) methods of type classes. Parametric polymorphic functions behave the same at all type instances, whereas class methods can change their behaviour in dependence on the class type parameters. Similarly, vanilla type constructors imply the same data representation for all type instances, but family constructors can have varying representation types for varying type indices.

Indexed type families come in three flavours: data families, open type synonym families, and closed type synonym families. They are the indexed family variants of algebraic data types and type synonyms, respectively. The instances of data families can be data types and newtypes.

Type families are enabled by the language extension *`TypeFamilies`* (page 338). Additional information on the use of type families in GHC is available on the Haskell wiki page on type families.

6.4.9.1 Data families

Data families appear in two flavours: (1) they can be defined on the toplevel or (2) they can appear inside type classes (in which case they are known as associated types). The former is the more general variant, as it lacks the requirement for the type-indexes to coincide with the class parameters. However, the latter can lead to more clearly structured code and compiler warnings if some type instances were - possibly accidentally - omitted. In the following, we always discuss the general toplevel form first and then cover the additional constraints placed on associated types.

Data family declarations

Indexed data families are introduced by a signature, such as

```
data family GMap k :: Type -> Type
```

The special `family` distinguishes family from standard data declarations. The result kind annotation is optional and, as usual, defaults to `Type` if omitted. An example is

```
data family Array e
```

Named arguments can also be given explicit kind signatures if needed. Just as with *GADT declarations* (page 335) named arguments are entirely optional, so that we can declare `Array` alternatively with

```
data family Array :: Type -> Type
```

Unlike with ordinary data definitions, the result kind of a data family does not need to be `Type`. It can alternatively be:

- Of the form TYPE r for some r (see *Representation polymorphism* (page 381)). For example:

```
data family DF1 :: TYPE IntRep
data family DF2 (r :: RuntimeRep)  :: TYPE r
data family DF3 :: Type -> TYPE WordRep
```

- A bare kind variable (with *PolyKinds* (page 364) enabled). For example:

```
data family DF4 :: k
data family DF5 (a :: k) :: k
data family DF6 :: (k -> Type) -> k
```

Data instances' kinds must end in Type, however. This restriction is slightly relaxed when the *UnliftedNewtypes* (page 558) extension is enabled, as it permits a newtype instance's kind to end in TYPE r for some r.

Data instance declarations

Instance declarations of data and newtype families are very similar to standard data and newtype declarations. The only two differences are that the keyword data or newtype is followed by instance and that some or all of the type arguments can be non-variable types, but may not contain forall types or type synonym families. However, data families are generally allowed in type parameters, and type synonyms are allowed as long as they are fully applied and expand to a type that is itself admissible - exactly as this is required for occurrences of type synonyms in class instance parameters. For example, the Either instance for GMap is

```
data instance GMap (Either a b) v = GMapEither (GMap a v) (GMap b v)
```

In this example, the declaration has only one variant. In general, it can be any number.

When *ExplicitForAll* (page 506) is enabled, type and kind variables used on the left hand side can be explicitly bound. For example:

```
data instance forall a (b :: Proxy a). F (Proxy b) = FProxy Bool
```

When an explicit forall is present, *all* type and kind variables mentioned which are not already in scope must be bound by the forall:

```
data instance forall   (a :: k). F a = FOtherwise  -- rejected: k not in scope
data instance forall k (a :: k). F a = FOtherwise  -- accepted
```

When the flag *-Wunused-type-patterns* (page 104) is enabled, type variables that are mentioned in the patterns on the left hand side, but not used on the right hand side are reported. Variables that occur multiple times on the left hand side are also considered used. To suppress the warnings, unused variables should be either replaced or prefixed with underscores. Type variables starting with an underscore (_x) are otherwise treated as ordinary type variables.

This resembles the wildcards that can be used in *Partial Type Signatures* (page 520). However, there are some differences. No error messages reporting the inferred types are generated, nor does the extension *PartialTypeSignatures* (page 520) have any effect.

A type or kind variable explicitly bound using *ExplicitForAll* (page 506) but not used on the left hand side will generate an error, not a warning.

Data and newtype instance declarations are only permitted when an appropriate family declaration is in scope - just as a class instance declaration requires the class declaration to be visible. Moreover, each instance declaration has to conform to the kind determined by its family declaration. This implies that the number of parameters of an instance declaration matches the arity determined by the kind of the family.

A data family instance declaration can use the full expressiveness of ordinary `data` or `newtype` declarations:

- Although, a data family is *introduced* with the keyword "`data`", a data family *instance* can use either `data` or `newtype`. For example:

```
data family T a
data    instance T Int  = T1 Int | T2 Bool
newtype instance T Char = TC Bool
```

- A `data instance` can use GADT syntax for the data constructors, and indeed can define a GADT. For example:

```
data family G a b
data instance G [a] b where
   G1 :: c -> G [Int] b
   G2 :: G [a] Bool
```

- You can use a `deriving` clause on a `data instance` or `newtype instance` declaration.

Even if data families are defined as toplevel declarations, functions that perform different computations for different family instances may still need to be defined as methods of type classes. In particular, the following is not possible:

```
data family T a
data instance T Int  = A
data instance T Char = B
foo :: T a -> Int
foo A = 1
foo B = 2
```

Instead, you would have to write `foo` as a class operation, thus:

```
class Foo a where
  foo :: T a -> Int
instance Foo Int where
  foo A = 1
instance Foo Char where
  foo B = 2
```

Given the functionality provided by GADTs (Generalised Algebraic Data Types), it might seem as if a definition, such as the above, should be feasible. However, type families - in contrast to GADTs - are *open;* i.e., new instances can always be added, possibly in other modules. Supporting pattern matching across different data instances would require a form of extensible case construct.

Overlap of data instances

The instance declarations of a data family used in a single program may not overlap at all, independent of whether they are associated or not. In contrast to type class instances, this is not only a matter of consistency, but one of type safety.

6.4.9.2 Synonym families

Type families appear in three flavours: (1) they can be defined as open families on the toplevel, (2) they can be defined as closed families on the toplevel, or (3) they can appear inside type classes (in which case they are known as associated type synonyms). Toplevel families are more general, as they lack the requirement for the type-indexes to coincide with the class parameters. However, associated type synonyms can lead to more clearly structured code and compiler warnings if some type instances were - possibly accidentally - omitted. In the following, we always discuss the general toplevel forms first and then cover the additional constraints placed on associated types. Note that closed associated type synonyms do not exist.

Type family declarations

Open indexed type families are introduced by a signature, such as

```
type family Elem c :: Type
```

The special `family` distinguishes family from standard type declarations. The result kind annotation is optional and, as usual, defaults to `Type` if omitted. An example is

```
type family Elem c
```

Parameters can also be given explicit kind signatures if needed. We call the number of parameters in a type family declaration, the family's arity, and all applications of a type family must be fully saturated with respect to that arity. This requirement is unlike ordinary type synonyms and it implies that the kind of a type family is not sufficient to determine a family's arity, and hence in general, also insufficient to determine whether a type family application is well formed. As an example, consider the following declaration:

```
type family F a b :: Type -> Type
  -- F's arity is 2,
  -- although its overall kind is Type -> Type -> Type -> Type
```

Given this declaration the following are examples of well-formed and malformed types:

```
F Char [Int]       -- OK!  Kind: Type -> Type
F Char [Int] Bool  -- OK!  Kind: Type
F IO Bool          -- WRONG: kind mismatch in the first argument
F Bool             -- WRONG: unsaturated application
```

The result kind annotation is optional and defaults to `Type` (like argument kinds) if omitted. Polykinded type families can be declared using a parameter in the kind annotation:

```
type family F a :: k
```

In this case the kind parameter `k` is actually an implicit parameter of the type family.

At definition site, the arity determines what inputs can be matched on:

```
data PT (a :: Type)

type family F1 :: k -> Type
type instance F1 = PT
  -- OK, 'k' can be matched on.

type family F0 :: forall k. k -> Type
type instance F0 = PT
  -- Error:
  --   • Expected kind ‘forall k. k -> Type’,
  --       but ‘PT’ has kind ‘Type -> Type’
  --   • In the type ‘PT’
  --     In the type instance declaration for ‘F0’
```

Both F1 and F0 have kind `forall k. k -> Type`, but their arity differs.

At use sites, the arity determines if the definition can be used in a higher-rank scenario:

```
type HRK (f :: forall k. k -> Type) = (f Int, f Maybe, f True)

type H1 = HRK F0  -- OK
type H2 = HRK F1
  -- Error:
  --   • Expected kind ‘forall k. k -> Type’,
  --       but ‘F1’ has kind ‘k0 -> Type’
  --   • In the first argument of ‘HRK’, namely ‘F1’
  --     In the type ‘HRK F1’
  --     In the type declaration for ‘H2’
```

This is a consequence of the requirement that all applications of a type family must be fully saturated with respect to their arity.

Type instance declarations

Instance declarations of type families are very similar to standard type synonym declarations. The only two differences are that the keyword `type` is followed by `instance` and that some or all of the type arguments can be non-variable types, but may not contain forall types or type synonym families. However, data families are generally allowed, and type synonyms are allowed as long as they are fully applied and expand to a type that is admissible - these are the exact same requirements as for data instances. For example, the `[e]` instance for `Elem` is

```
type instance Elem [e] = e
```

Type arguments can be replaced with underscores (_) if the names of the arguments don't matter. This is the same as writing type variables with unique names. Unused type arguments can be replaced or prefixed with underscores to avoid warnings when the *-Wunused-type-patterns* (page 104) flag is enabled. The same rules apply as for *Data instance declarations* (page 340).

Also in the same way as *Data instance declarations* (page 340), when *ExplicitForAll* (page 506) is enabled, type and kind variables can be explicitly bound in a type instance declaration.

Type family instance declarations are only legitimate when an appropriate family declaration is in scope - just like class instances require the class declaration to be visible. Moreover, each instance declaration has to conform to the kind determined by its family declaration, and the number of type parameters in an instance declaration must match the number of type parameters in the family declaration. Finally, the right-hand side of a type instance must be a monotype (i.e., it may not include foralls) and after the expansion of all saturated vanilla type synonyms, no synonyms, except family synonyms may remain.

Closed type families

A type family can also be declared with a `where` clause, defining the full set of equations for that family. For example:

```
type family F a where
  F Int  = Double
  F Bool = Char
  F a    = String
```

A closed type family's equations are tried in order, from top to bottom, when simplifying a type family application. In this example, we declare an instance for `F` such that `F Int` simplifies to `Double`, `F Bool` simplifies to `Char`, and for any other type `a` that is known not to be `Int` or `Bool`, `F a` simplifies to `String`. Note that GHC must be sure that `a` cannot unify with `Int` or `Bool` in that last case; if a programmer specifies just `F a` in their code, GHC will not be able to simplify the type. After all, `a` might later be instantiated with `Int`.

A closed type family's equations have the same restrictions and extensions as the equations for open type family instances. For instance, when *ExplicitForAll* (page 506) is enabled, type or kind variables used on the left hand side of an equation can be explicitly bound, such as in:

```
type family R a where
  forall t a. R (t a) = [a]
  forall a.   R a     = a
```

A closed type family may be declared with no equations. Such closed type families are opaque type-level definitions that will never reduce, are not necessarily injective (unlike empty data types), and cannot be given any instances. This is different from omitting the equations of a closed type family in a `hs-boot` file, which uses the syntax `where ..`, as in that case there may or may not be equations given in the `hs` file.

Type family examples

Here are some examples of admissible and illegal type instances:

```
type family F a :: Type
type instance F [Int]   = Int   -- OK!
type instance F String  = Char  -- OK!
type instance F (F a)   = a     -- WRONG: type parameter mentions a type family
type instance
  F (forall a. (a, b))  = b     -- WRONG: a forall type appears in a type parameter
type instance
  F Float = forall a.a          -- WRONG: right-hand side may not be a forall type
type family H a where          -- OK!
  H Int  = Int
  H Bool = Bool
  H a    = String
type instance H Char = Char    -- WRONG: cannot have instances of closed family
```

(continues on next page)

(continued from previous page)

```
type family K a where         -- OK!

type family G a b :: Type -> Type
type instance G Int            = (,)     -- WRONG: must be two type parameters
type instance G Int Char Float = Double  -- WRONG: must be two type parameters
```

Compatibility and apartness of type family equations

There must be some restrictions on the equations of type families, lest we define an ambiguous rewrite system. So, equations of open type families are restricted to be compatible. Two type patterns are compatible if

1. all corresponding types and implicit kinds in the patterns are apart, or
2. the two patterns unify producing a substitution, and the right-hand sides are equal under that substitution.

Two types are considered apart if, for all possible substitutions, the types cannot reduce to a common reduct.

The first clause of "compatible" is the more straightforward one. It says that the patterns of two distinct type family instances cannot overlap. For example, the following is disallowed:

```
type instance F Int = Bool
type instance F Int = Char
```

The second clause is a little more interesting. It says that two overlapping type family instances are allowed if the right-hand sides coincide in the region of overlap. Some examples help here:

```
type instance F (a, Int) = [a]
type instance F (Int, b) = [b]   -- overlap permitted

type instance G (a, Int)  = [a]
type instance G (Char, a) = [a]  -- ILLEGAL overlap, as [Char] /= [Int]
```

Note that this compatibility condition is independent of whether the type family is associated or not, and it is not only a matter of consistency, but one of type safety.

For a polykinded type family, the kinds are checked for apartness just like types. For example, the following is accepted:

```
type family J a :: k
type instance J Int = Bool
type instance J Int = Maybe
```

These instances are compatible because they differ in their implicit kind parameter; the first uses `Type` while the second uses `Type -> Type`.

The definition for "compatible" uses a notion of "apart", whose definition in turn relies on type family reduction. This condition of "apartness", as stated, is impossible to check, so we use this conservative approximation: two types are considered to be apart when the two types cannot be unified, even by a potentially infinite unifier. Allowing the unifier to be infinite disallows the following pair of instances:

```
type instance H x   x = Int
type instance H [x] x = Bool
```

The type patterns in this pair equal if x is replaced by an infinite nesting of lists. Rejecting instances such as these is necessary for type soundness.

Compatibility also affects closed type families. When simplifying an application of a closed type family, GHC will select an equation only when it is sure that no incompatible previous equation will ever apply. Here are some examples:

```
type family F a where
  F Int = Bool
  F a   = Char

type family G a where
  G Int = Int
  G a   = a
```

In the definition for `F`, the two equations are incompatible – their patterns are not apart, and yet their right-hand sides do not coincide. Thus, before GHC selects the second equation, it must be sure that the first can never apply. So, the type `F a` does not simplify; only a type such as `F Double` will simplify to `Char`. In `G`, on the other hand, the two equations are compatible. Thus, GHC can ignore the first equation when looking at the second. So, `G a` will simplify to `a`.

Incompatibilities between closed type family equations can be displayed in *:info* (page 48) when *-fprint-axiom-incomps* (page 77) is enabled.

However see *Type, class and other declarations* (page 23) for the overlap rules in GHCi.

Decidability of type synonym instances

In order to guarantee that type inference in the presence of type families is decidable, we need to place a number of additional restrictions on the formation of type instance declarations (c.f., Definition 5 (Relaxed Conditions) of "Type Checking with Open Type Functions"). Instance declarations have the general form

```
type instance F t1 .. tn = t
```

where we require that for every type family application (`G s1 .. sm`) in `t`,

1. `s1 .. sm` do not contain any type family constructors,
2. the total number of symbols (data type constructors and type variables) in `s1 .. sm` is strictly smaller than in `t1 .. tn`, and
3. for every type variable `a`, `a` occurs in `s1 .. sm` at most as often as in `t1 .. tn`.

These restrictions are easily verified and ensure termination of type inference. However, they are not sufficient to guarantee completeness of type inference in the presence of, so called, ''loopy equalities'', such as `a ~ [F a]`, where a recursive occurrence of a type variable is underneath a family application and data constructor application - see the above mentioned paper for details.

If the option *UndecidableInstances* (page 484) is passed to the compiler (see *Undecidable instances and loopy superclasses* (page 484)), the above restrictions are not enforced and it is on the programmer to ensure termination of the normalisation of type families during type inference.

Reducing type family applications

-ffamily-application-cache

The flag *-ffamily-application-cache* (page 347) (on by default) instructs GHC to use a cache when reducing type family applications. In most cases, this will speed up compilation. The use of this flag will not affect runtime behaviour.

When GHC encounters a type family application (like `F Int a`) in a program, it must often reduce it in order to complete type checking. Here is a simple example:

```
type family F a where
  F Int            = Bool
  F (Maybe Double) = Char

g :: F Int -> Bool
g = not
```

Despite the fact that `g`'s type mentions `F Int`, GHC must recognize that `g`'s argument really has type `Bool`. This is done by *reducing* `F Int` to become `Bool`. Sometimes, there is not enough information to reduce a type family application; we say such an application is *stuck*. Continuing this example, an occurrence of `F (Maybe a)` (for some type variable `a`) would be stuck, as no equation applies.

During type checking, GHC uses heuristics to determine which type family application to reduce next; there is no predictable ordering among different type family applications. The non-determinism rarely matters in practice. In most programs, type family reduction terminates, and so these choices are immaterial. However, if a type family application does not terminate, it is possible that type-checking may unpredictably diverge. (GHC will always take the same path for a given source program, but small changes in that source program may induce GHC to take a different path. Compiling a given, unchanged source program is still deterministic.)

In order to speed up type family reduction, GHC normally uses a cache, remembering what type family applications it has previously reduced. This feature can be disabled with *-fno-family-application-cache* (page 347).

6.4.9.3 Wildcards on the LHS of data and type family instances

When the name of a type argument of a data or type instance declaration doesn't matter, it can be replaced with an underscore (_). This is the same as writing a type variable with a unique name.

```
data family F a b :: Type
data instance F Int _ = Int
-- Equivalent to  data instance F Int b = Int

type family T a :: Type
type instance T (a,_) = a
-- Equivalent to  type instance T (a,b) = a
```

This use of underscore for wildcard in a type pattern is exactly like pattern matching in the term language, but is rather different to the use of a underscore in a partial type signature (see *Type Wildcards* (page 521)).

A type variable beginning with an underscore is not treated specially in a type or data instance declaration. For example:

```
data instance F Bool _a = _a -> Int
-- Equivalent to  data instance F Bool a = a -> Int
```

Contrast this with the special treatment of named wildcards in type signatures (*Named Wildcards* (page 522)).

6.4.9.4 Associated data and type families

A data or type synonym family can be declared as part of a type class, thus:

```
class GMapKey k where
  data GMap k :: Type -> Type
  ...

class Collects ce where
  type Elem ce :: Type
  ...
```

When doing so, we (optionally) may drop the "family" keyword.

The type parameters must all be type variables, of course, and some (but not necessarily all) of then can be the class parameters. Each class parameter may only be used at most once per associated type, but some may be omitted and they may be in an order other than in the class head. Hence, the following contrived example is admissible:

```
class C a b c where
  type T c a x :: Type
```

Here c and a are class parameters, but the type is also indexed on a third parameter x.

Associated instances

When an associated data or type synonym family instance is declared within a type class instance, we (optionally) may drop the instance keyword in the family instance:

```
instance (GMapKey a, GMapKey b) => GMapKey (Either a b) where
  data GMap (Either a b) v = GMapEither (GMap a v) (GMap b v)
  ...

instance Eq (Elem [e]) => Collects [e] where
  type Elem [e] = e
  ...
```

The data or type family instance for an associated type must follow the rule that the type indexes corresponding to class parameters must be precisely the same as types given in the instance head. For example:

```
class Collects ce where
  type Elem ce :: Type

instance Eq (Elem [e]) => Collects [e] where
  -- Choose one of the following alternatives:
  type Elem [e] = e       -- OK
  type Elem [x] = x       -- BAD; '[x]' is different to '[e]' from head
```

(continues on next page)

(continued from previous page)

```
type Elem x   = x       -- BAD; 'x' is different to '[e]'
type Elem [Maybe x] = x -- BAD: '[Maybe x]' is different to '[e]'
```

Note the following points:

- An instance for an associated family can only appear as part of an instance declarations of the class in which the family was declared, just as with the equations of the methods of a class.
- The type variables on the right hand side of the type family equation must, as usual, be explicitly bound by the left hand side. This restriction is relaxed for *kind* variables, however, as the right hand side is allowed to mention kind variables that are implicitly bound. For example, these are legitimate:

```
data family Nat :: k -> k -> Type
-- k is implicitly bound by an invisible kind pattern
newtype instance Nat :: (k -> Type) -> (k -> Type) -> Type where
  Nat :: (forall xx. f xx -> g xx) -> Nat f g

class Funct f where
  type Codomain f :: Type
instance Funct ('KProxy :: KProxy o) where
  -- o is implicitly bound by the kind signature
  -- of the LHS type pattern ('KProxy)
  type Codomain 'KProxy = NatTr (Proxy :: o -> Type)
```

- The instance for an associated type can be omitted in class instances. In that case, unless there is a default instance (see *Associated type synonym defaults* (page 350)), the corresponding instance type is not inhabited; i.e., only diverging expressions, such as `undefined`, can assume the type.
- Although it is unusual, there (currently) can be *multiple* instances for an associated family in a single instance declaration. For example, this is legitimate:

```
instance GMapKey Flob where
  data GMap Flob [v] = G1 v
  data GMap Flob Int = G2 Int
  ...
```

  Here we give two data instance declarations, one in which the last parameter is `[v]`, and one for which it is `Int`. Since you cannot give any *subsequent* instances for `(GMap Flob ...)`, this facility is most useful when the free indexed parameter is of a kind with a finite number of alternatives (unlike `Type`).

- When *ExplicitForAll* (page 506) is enabled, type and kind variables can be explicitly bound in associated data or type family instances in the same way (and with the same restrictions) as *Data instance declarations* (page 340) or *Type instance declarations* (page 343). For example, adapting the above, the following is accepted:

```
instance Eq (Elem [e]) => Collects [e] where
  type forall e. Elem [e] = e
```

Associated type synonym defaults

It is possible for the class defining the associated type to specify a default for associated type instances. So for example, this is OK:

```
class IsBoolMap v where
  type Key v
  type instance Key v = Int

  lookupKey :: Key v -> v -> Maybe Bool

instance IsBoolMap [(Int, Bool)] where
  lookupKey = lookup
```

In an `instance` declaration for the class, if no explicit `type instance` declaration is given for the associated type, the default declaration is used instead, just as with default class methods.

Note the following points:

- The `instance` keyword is optional.
- There can be at most one default declaration for an associated type synonym.
- A default declaration is not permitted for an associated *data* type.
- The default declaration must mention only type *variables* on the left hand side, and type variables may not be repeated on the left-hand side. The right hand side must mention only type variables that are explicitly bound on the left hand side. This restriction is relaxed for *kind* variables, however, as the right hand side is allowed to mention kind variables that are implicitly bound on the left hand side.

  Like with *Associated instances* (page 348), it is possible to explicitly bind type and kind variables in default declarations with a `forall` by using the *`ExplicitForAll`* (page 506) language extension.
- Unlike the associated type family declaration itself, the type variables of the default instance are independent of those of the parent class.

Here are some examples:

```
class C (a :: Type) where
  type F1 a :: Type
  type instance F1 a = [a]     -- OK
  type instance F1 a = a->a    -- BAD; only one default instance is allowed

  type F2 b a                  -- OK; note the family has more type
                               --     variables than the class
  type instance F2 c d = c->d  -- OK; you don't have to use 'a' in the type instance

  type F3 a
  type F3 [b] = b              -- BAD; only type variables allowed on the
                               --      LHS, and the argument to F3 is
                               --      instantiated to [b], which is not
                               --      a bare type variable

  type F4 x y
  type F4 x x = x              -- BAD; the type variable x is repeated on
                               --      the LHS
```

(continues on next page)

(continued from previous page)

```
type F5 a
type F5 b = a                   -- BAD; 'a' is not in scope  in the RHS

type F6 a :: [k]
type F6 a = ('[] :: [x])        -- OK; the kind variable x is implicitly
                                --     bound by an invisible kind pattern
                                --     on the LHS

type F7 a
type F7 a =
  Proxy ('[] :: [x])            -- BAD; the kind variable x is not bound,
                                --      even by an invisible kind pattern

type F8 (x :: a) :: [a]
type F8 x = ('[] :: [a])        -- OK; the kind variable a is implicitly
                                --     bound by the kind signature of the
                                --     LHS type pattern

type F9 (a :: k)
type F9 a = Maybe a             -- BAD; the kind variable k is
                                --      instantiated to Type, which is not
                                --      a bare kind variable

type F10 (a :: j) (b :: k)
type F10 (a :: z) (b :: z)
  = Proxy a                     -- BAD; the kind variable z is repeated,
                                --      as both j and k are instantiated to z

type F11 a b
type forall a b. F11 a b = a -- OK; LHS type variables can be
                                --     explicitly bound with 'forall'

type F12 (a :: k)
type F12 @k a = Proxy a         -- OK; visible kind application syntax is
                                --     permitted in default declarations
```

Scoping of class parameters

The visibility of class parameters in the right-hand side of associated family instances depends *solely* on the parameters of the family. As an example, consider the simple class declaration

```
class C a b where
  data T a
```

Only one of the two class parameters is a parameter to the data family. Hence, the following instance declaration is invalid:

```
instance C [c] d where
  data T [c] = MkT (c, d)    -- WRONG!!  'd' is not in scope
```

Here, the right-hand side of the data instance mentions the type variable `d` that does not occur in its left-hand side. We cannot admit such data instances as they would compromise type safety.

Bear in mind that it is also possible for the *right*-hand side of an associated family instance to contain *kind* parameters (by using the *PolyKinds* (page 364) extension). For instance, this class and instance are perfectly admissible:

```
class C k where
  type T :: k

instance C (Maybe a) where
  type T = (Nothing :: Maybe a)
```

Here, although the right-hand side `(Nothing :: Maybe a)` mentions a kind variable `a` which does not occur on the left-hand side, this is acceptable, because `a` is *implicitly* bound by `T`'s kind pattern.

A kind variable can also be bound implicitly in a LHS type pattern, as in this example:

```
class C a where
  type T (x :: a) :: [a]

instance C (Maybe a) where
  type T x = ('[] :: [Maybe a])
```

In `('[] :: [Maybe a])`, the kind variable `a` is implicitly bound by the kind signature of the LHS type pattern `x`.

Instance contexts and associated type and data instances

Associated type and data instance declarations do not inherit any context specified on the enclosing instance. For type instance declarations, it is unclear what the context would mean. For data instance declarations, it is unlikely a user would want the context repeated for every data constructor. The only place where the context might likely be useful is in a `deriving` clause of an associated data instance. However, even here, the role of the outer instance context is murky. So, for clarity, we just stick to the rule above: the enclosing instance context is ignored. If you need to use a non-trivial context on a derived instance, use a *standalone deriving* (page 436) clause (at the top level).

6.4.9.5 Import and export

The rules for export lists (Haskell Report Section 5.2) need adjustment for type families:

- The form `T(..)`, where `T` is a data family, names the family `T` and all the in-scope constructors (whether in scope qualified or unqualified) that are data instances of `T`.
- The form `T(.., ci, .., fj, ..)`, where `T` is a data family, names `T` and the specified constructors `ci` and fields `fj` as usual. The constructors and field names must belong to some data instance of `T`, but are not required to belong to the *same* instance.
- The form `C(..)`, where `C` is a class, names the class `C` and all its methods *and associated types*.
- The form `C(.., mi, .., type Tj, ..)`, where `C` is a class, names the class `C`, and the specified methods `mi` and associated types `Tj`. The types need a keyword "`type`" to distinguish them from data constructors.
- Whenever there is no export list and a data instance is defined, the corresponding data family type constructor is exported along with the new data constructors, regardless of whether the data family is defined locally or in another module.

Examples

Recall our running GMapKey class example:

```
class GMapKey k where
  data GMap k :: Type -> Type
  insert :: GMap k v -> k -> v -> GMap k v
  lookup :: GMap k v -> k -> Maybe v
  empty  :: GMap k v

instance (GMapKey a, GMapKey b) => GMapKey (Either a b) where
  data GMap (Either a b) v = GMapEither (GMap a v) (GMap b v)
  ...method declarations...
```

Here are some export lists and their meaning:

- ```
  module GMap( GMapKey )
  ```

  Exports just the class name.

- ```
  module GMap( GMapKey(..) )
  ```

  Exports the class, the associated type `GMap` and the member functions `empty`, `lookup`, and `insert`. The data constructors of `GMap` (in this case `GMapEither`) are not exported.

- ```
  module GMap( GMapKey( type GMap, empty, lookup, insert ) )
  ```

  Same as the previous item. Note the "`type`" keyword.

- ```
  module GMap( GMapKey(..), GMap(..) )
  ```

  Same as previous item, but also exports all the data constructors for `GMap`, namely `GMapEither`.

- ```
  module GMap ( GMapKey( empty, lookup, insert), GMap(..) )
  ```

  Same as previous item.

- ```
  module GMap ( GMapKey, empty, lookup, insert, GMap(..) )
  ```

  Same as previous item.

Two things to watch out for:

- You cannot write `GMapKey(type GMap(..))` — i.e., sub-component specifications cannot be nested. To specify `GMap`'s data constructors, you have to list it separately.
- Consider this example:

  ```
  module X where
    data family D

  module Y where
    import X
    data instance D Int = D1 | D2
  ```

  Module `Y` exports all the entities defined in `Y`, namely the data constructors `D1` and `D2`, and *implicitly* the data family `D`, even though it's defined in `X`. This means you can write `import Y( D(D1,D2) )` *without* giving an explicit export list like this:

```
     module Y( D(..) ) where ...
or   module Y( module Y, D ) where ...
```

Instances

Family instances are implicitly exported, just like class instances. However, this applies only to the heads of instances, not to the data constructors an instance defines.

6.4.9.6 Type families and instance declarations

Type families require us to extend the rules for the form of instance heads, which are given in *Relaxed rules for the instance head* (page 480). Specifically:

- Data type families may appear in an instance head
- Type synonym families may not appear (at all) in an instance head

The reason for the latter restriction is that there is no way to check for instance matching. Consider

```
type family F a
type instance F Bool = Int

class C a

instance C Int
instance C (F a)
```

Now a constraint `(C (F Bool))` would match both instances. The situation is especially bad because the type instance for `F Bool` might be in another module, or even in a module that is not yet written.

However, type class instances of instances of data families can be defined much like any other data type. For example, we can say

```
data instance T Int = T1 Int | T2 Bool
instance Eq (T Int) where
  (T1 i) == (T1 j) = i==j
  (T2 i) == (T2 j) = i==j
  _      == _      = False
```

Note that class instances are always for particular *instances* of a data family and never for an entire family as a whole. This is for essentially the same reasons that we cannot define a toplevel function that performs pattern matching on the data constructors of *different* instances of a single type family. It would require a form of extensible case construct.

Data instance declarations can also have `deriving` clauses. For example, we can write

```
data GMap () v = GMapUnit (Maybe v)
                 deriving Show
```

which implicitly defines an instance of the form

```
instance Show v => Show (GMap () v) where ...
```

6.4.9.7 Injective type families

TypeFamilyDependencies

Implies *TypeFamilies* (page 338)

Since 8.0.1

Allow functional dependency annotations on type families. This allows one to define injective type families.

Starting with GHC 8.0 type families can be annotated with injectivity information. This information is then used by GHC during type checking to resolve type ambiguities in situations where a type variable appears only under type family applications. Consider this contrived example:

```
type family Id a
type instance Id Int = Int
type instance Id Bool = Bool

id :: Id t -> Id t
id x = x
```

Here the definition of `id` will be rejected because type variable `t` appears only under type family applications and is thus ambiguous. But this code will be accepted if we tell GHC that `Id` is injective, which means it will be possible to infer `t` at call sites from the type of the argument:

```
type family Id a = r | r -> a
```

Injective type families are enabled with `-XTypeFamilyDependencies` language extension. This extension implies `-XTypeFamilies`.

For full details on injective type families refer to the Haskell Symposium 2015 paper Injective type families for Haskell.

Syntax of injectivity annotation

The injectivity annotation is added after the type family head and consists of two parts:

- a type variable that names the result of a type family. Syntax: `= tyvar` or `= (tyvar :: kind)`. The type variable must be fresh.
- an injectivity annotation of the form `| A -> B`, where `A` is the result type variable (see previous bullet) and `B` is a list of argument type and kind variables in which type family is injective. It is possible to omit some variables if the type family is not injective in them.

Examples:

```
type family Id a = result | result -> a where
type family F a b c = d | d -> a c b
type family G (a :: k) b c = foo | foo -> k b where
```

For open and closed type families it is OK to name the result but skip the injectivity annotation. This is not the case for associated type synonyms, where the named result without injectivity annotation will be interpreted as associated type synonym default.

Verifying the injectivity annotation against type family equations

Once the user declares type family to be injective GHC must verify that this declaration is correct, i.e., that type family equations don't violate the injectivity annotation. A general idea is that if at least one equation (bullets (1), (2) and (3) below) or a pair of equations (bullets (4) and (5) below) violates the injectivity annotation then a type family is not injective in a way the user claims and an error is reported. In the bullets below *RHS* refers to the right-hand side of the type family equation being checked for injectivity. *LHS* refers to the arguments of that type family equation. Below are the rules followed when checking injectivity of a type family:

1. If a RHS of a type family equation is a type family application GHC reports that the type family is not injective.
2. If a RHS of a type family equation is a bare type variable we require that all LHS variables (including implicit kind variables) are also bare. In other words, this has to be a sole equation of that type family and it has to cover all possible patterns. If the patterns are not covering GHC reports that the type family is not injective.
3. If a LHS type variable that is declared as injective is not mentioned on injective position in the RHS GHC reports that the type family is not injective. Injective position means either argument to a type constructor or injective argument to a type family. Type inference can potentially loop when looking under injective type families in the RHS, so this requires `UndecidableInstances` (page 484); GHC suggests enabling the flag when it is necessary.
4. *Open type families* Open type families are typechecked incrementally. This means that when a module is imported type family instances contained in that module are checked against instances present in already imported modules.

   A pair of an open type family equations is checked by attempting to unify their RHSs. If the RHSs don't unify this pair does not violate injectivity annotation. If unification succeeds with a substitution then LHSs of unified equations must be identical under that substitution. If they are not identical then GHC reports that the type family is not injective.
5. In a *closed type family* all equations are ordered and in one place. Equations are also checked pair-wise but this time an equation has to be paired with all the preceding equations. Of course a single-equation closed type family is trivially injective (unless (1), (2) or (3) above holds).

   When checking a pair of closed type family equations GHC tried to unify their RHSs. If they don't unify this pair of equations does not violate injectivity annotation. If the RHSs can be unified under some substitution (possibly empty) then either the LHSs unify under the same substitution or the LHS of the latter equation is subsumed by earlier equations. If neither condition is met GHC reports that a type family is not injective.

Note that for the purpose of injectivity check in bullets (4) and (5) GHC uses a special variant of unification algorithm that treats type family applications as possibly unifying with anything.

6.4.10 Datatype promotion

DataKinds

Since 7.4.1

Status Included in *GHC2024* (page 270)

Allow promotion of data types to kind level.

This section describes *data type promotion*, an extension to the kind system that complements kind polymorphism. It is enabled by *DataKinds* (page 357), and described in more detail in the paper Giving Haskell a Promotion, which appeared at TLDI 2012. See also *TypeData* (page 363) for a more fine-grained alternative.

6.4.10.1 Motivation

Standard Haskell has a rich type language. Types classify terms and serve to avoid many common programming mistakes. The kind language, however, is relatively simple, distinguishing only regular types (kind Type) and type constructors (e.g. kind Type -> Type -> Type). In particular when using advanced type system features, such as type families (*Type families* (page 338)) or GADTs (*Generalised Algebraic Data Types (GADTs)* (page 335)), this simple kind system is insufficient, and fails to prevent simple errors. Consider the example of type-level natural numbers, and length-indexed vectors:

```
data Ze
data Su n

data Vec :: Type -> Type -> Type where
  Nil  :: Vec a Ze
  Cons :: a -> Vec a n -> Vec a (Su n)
```

The kind of Vec is Type -> Type -> Type. This means that, e.g., Vec Int Char is a well-kinded type, even though this is not what we intend when defining length-indexed vectors.

With *DataKinds* (page 357), the example above can then be rewritten to:

```
data Nat = Ze | Su Nat

data Vec :: Type -> Nat -> Type where
  Nil  :: Vec a Ze
  Cons :: a -> Vec a n -> Vec a (Su n)
```

With the improved kind of Vec, things like Vec Int Char are now ill-kinded, and GHC will report an error.

6.4.10.2 Overview

With *DataKinds* (page 357), GHC automatically promotes every datatype to be a kind and its (value) constructors to be type constructors. The following types

```
data Nat = Zero | Succ Nat

data List a = Nil | Cons a (List a)

data Pair a b = MkPair a b
```

(continues on next page)

(continued from previous page)

```
data Sum a b = L a | R b
```

give rise to the following kinds and type constructors:

```
Nat :: Type
Zero :: Nat
Succ :: Nat -> Nat

List :: Type -> Type
Nil  :: forall k. List k
Cons :: forall k. k -> List k -> List k

Pair  :: Type -> Type -> Type
MkPair :: forall k1 k2. k1 -> k2 -> Pair k1 k2

Sum :: Type -> Type -> Type
L :: k1 -> Sum k1 k2
R :: k2 -> Sum k1 k2
```

Virtually all data constructors, even those with rich kinds, can be promoted. There are only a couple of exceptions to this rule:

- Data family instance constructors cannot be promoted at the moment. GHC’s type theory just isn’t up to the task of promoting data families, which requires full dependent types.
- Data constructors with contexts cannot be promoted. For example:

```
data Foo :: Type -> Type where
  MkFoo :: Show a => Foo a    -- not promotable
```

The following kinds and promoted data constructors can be used even when *DataKinds* (page 357) is not enabled:

- `Type`
- `TYPE` (see *Representation polymorphism* (page 381))
- `Constraint` (see *The Constraint kind* (page 500))
- `CONSTRAINT`
- `Multiplicity` and its promoted data constructors (see *LinearTypes* (page 408))
- `LiftedRep` (see *Representation polymorphism* (page 381))
- `RuntimeRep` and its promoted data constructors (see *Representation polymorphism* (page 381))
- `Levity` and its promoted data constructors (see *Representation polymorphism* (page 381))
- `VecCount` and its promoted data constructors
- `VecElem` and its promoted data constructors

It is also possible to use kinds declared with `type data` (see *TypeData* (page 363)) without enabling *DataKinds* (page 357).

6.4.10.3 Distinguishing between types and constructors

Consider

```
data P = MkP    -- 1

data Prom = P   -- 2
```

The name `P` on the type level will refer to the type `P` (which has a constructor `MkP`) rather than the promoted data constructor `P` of kind `Prom`. To refer to the latter, prefix it with a single quote mark: `'P`.

This syntax can be used even if there is no ambiguity (i.e. there's no type `P` in scope).

GHC supports *-Wunticked-promoted-constructors* (page 102) that warns whenever a promoted data constructor is written without a quote mark. As of GHC 9.4, this warning is no longer enabled by *-Wall* (page 86); we no longer recommend quote marks as a preferred default (see #20531).

Just as in the case of Template Haskell (*Syntax* (page 528)), GHC gets confused if you put a quote mark before a data constructor whose second character is a quote mark. In this case, just put a space between the promotion quote and the data constructor:

```
data T = A'
type S = 'A'   -- ERROR: looks like a character
type R = ' A'  -- OK: promoted `A'`
```

6.4.10.4 Type-level literals

DataKinds (page 357) enables the use of numeric and string literals at the type level. For more information, see *Type-Level Literals* (page 384).

6.4.10.5 Promoted list and tuple types

With *DataKinds* (page 357), Haskell's list and tuple types are natively promoted to kinds, and enjoy the same convenient syntax at the type level, albeit prefixed with a quote:

```
data HList :: [Type] -> Type where
  HNil  :: HList '[]
  HCons :: a -> HList t -> HList (a ': t)

data Tuple :: (Type,Type) -> Type where
  Tuple :: a -> b -> Tuple '(a,b)

foo0 :: HList '[]
foo0 = HNil

foo1 :: HList '[Int]
foo1 = HCons (3::Int) HNil

foo2 :: HList [Int, Bool]
foo2 = ...
```

For type-level lists of *two or more elements*, such as the signature of `foo2` above, the quote may be omitted because the meaning is unambiguous. But for lists of one or zero elements (as in `foo0` and `foo1`), the quote is required, because the types `[]` and `[Int]` have existing meanings in Haskell.

Note: The declaration for `HCons` also requires *TypeOperators* (page 323) because of infix type operator `(':)`

6.4.10.6 Promoting existential data constructors

Note that we do promote existential data constructors that are otherwise suitable. For example, consider the following:

```
data Ex :: Type where
  MkEx :: forall a. a -> Ex
```

Both the type `Ex` and the data constructor `MkEx` get promoted, with the polymorphic kind `'MkEx :: forall k. k -> Ex`. Somewhat surprisingly, you can write a type family to extract the member of a type-level existential:

```
type family UnEx (ex :: Ex) :: k
type instance UnEx (MkEx x) = x
```

At first blush, `UnEx` seems poorly-kinded. The return kind `k` is not mentioned in the arguments, and thus it would seem that an instance would have to return a member of `k` *for any* `k`. However, this is not the case. The type family `UnEx` is a kind-indexed type family. The return kind `k` is an implicit parameter to `UnEx`. The elaborated definitions are as follows (where implicit parameters are denoted by braces):

```
type family UnEx {k :: Type} (ex :: Ex) :: k
type instance UnEx {k} (MkEx @k x) = x
```

Thus, the instance triggers only when the implicit parameter to `UnEx` matches the implicit parameter to `MkEx`. Because `k` is actually a parameter to `UnEx`, the kind is not escaping the existential, and the above code is valid.

See also #7347.

6.4.10.7 DataKinds and type synonyms

The *DataKinds* (page 357) extension interacts with type synonyms in the following ways:

1. In a *type* context: *DataKinds* (page 357) is not required to use a type synonym that expands to a type that would otherwise require the extension. For example:

```
{-# LANGUAGE DataKinds #-}
module A where

  type MyTrue = 'True

{-# LANGUAGE NoDataKinds #-}
module B where

  import A
  import Data.Proxy
```

(continues on next page)

(continued from previous page)

```
  f :: Proxy MyTrue
  f = Proxy
```

GHC will accept the type signature for `f` even though *DataKinds* (page 357) is not enabled, as the promoted data constructor `True` is tucked underneath the `MyTrue` type synonym. If the user had written `Proxy 'True` directly, however, then *DataKinds* (page 357) would be required.

2. In a *kind* context: *DataKinds* (page 357) applies to all types mentioned in the kind, *including the expansions of type synonyms*. For instance, given this module:

```
module C where

  type MyType = Type
  type MySymbol = Symbol
```

We would accept or reject the following definitions in this module, which makes use of *Standalone kind signatures and polymorphic recursion* (page 369):

```
{-# LANGUAGE NoDataKinds #-}
module D where

  import C

  -- ACCEPTED: The kind only mentions Type, which doesn't require DataKinds
  type D1 :: Type -> Type
  data D1 a

  -- REJECTED: The kind mentions Symbol, which requires DataKinds to use in
  -- a kind position
  data D2 :: Symbol -> Type
  data D2 a

  -- ACCEPTED: The kind mentions a type synonym MyType that expands to
  -- Type, which doesn't require DataKinds
  data D3 :: MyType -> Type
  data D3 a

  -- REJECTED: The kind mentions a type synonym MySymbol that expands to
  -- Symbol, which requires DataKinds to use in a kind position
  data D4 :: MySymbol -> Type
  data D4 a
```

6.4.11 Unique syntax for type-level lists and tuples

ListTuplePuns

Since 9.10.1

Accept bracket syntax to denote type constructors, using single quotes to disambiguate data constructors.

The previously defined mechanism for specifying data constructors with bracket syntax and single quotes is governed by this extension, which is enabled by default.

With NoListTuplePuns, brackets are unambiguously parsed as data constructors, while the single quote is not accepted as a prefix for them anymore. Type constructors cannot be expressed with brackets anymore; instead, new data type declarations in regular syntax have been added to ghc-prim:

```
data List a = [] | a : List a
data Unit = ()
data Tuple2 a b = (a, b)
data Tuple2# a b = (# a, b #)
data Sum2# a b = (# a | #) | (# | b #)
class (a, b) => CTuple2 a b
instance (c1, c2) => CTuple2 c1 c2
```

CTuple2 is a constraint tuple, which historically only concerns declarations like:

```
type C = (Eq Int, Ord String)
```

These are distinct from the usual specification of multiple constraints on functions or instances with parentheses, since those are treated specially by GHC.

When the extension is disabled, any occurrence of special syntax in types will be treated as the data constructor, so a type of `(Int, String)` has kind `Tuple2 Type Type`, corresponding to the type `'(Int, String)` with kind `(Type, Type)` when ListTuplePuns is enabled.

The explicit disambiguation syntax using single quotes is invalid syntax when the extension is disabled.

The earlier example would need to be rewritten like this:

```
data HList :: List Type -> Type where
  HNil  :: HList []
  HCons :: a -> HList t -> HList (a : t)

data Tuple :: Tuple2 Type Type -> Type where
  Tuple :: a -> b -> Tuple2 a b

foo0 :: HList []
foo0 = HNil

foo1 :: HList [Int]
foo1 = HCons (3 :: Int) HNil

foo2 :: HList [Int, Bool]
foo2 = ...
```

Constraint tuples may be mixed with conventional syntax:

```
f ::
  Monad m =>
  CTuple2 (Monad m) (Monad m) =>
  (Monad m, CTuple2 (Monad m) (Monad m)) =>
  m Int
f = pure 5
```

The new type constructors are exported only from the library ghc-experimental, by the modules Data.Tuple.Experimental, Data.Sum.Experimental and Prelude.Experimental.

Please refer to GHC Proposal #475 for the full specification of effects and interactions.

6.4.12 Type-level data declarations

TypeData

Since 9.6.1

Allow `type data` declarations, which define constructors at the type level.

This extension facilitates type-level (compile-time) programming by allowing a type-level counterpart of `data` declarations, such as this definition of type-level natural numbers:

```
type data Nat = Zero | Succ Nat
```

This is similar to the corresponding `data` declaration, except that the constructors `Zero` and `Succ` it introduces belong to the type constructor namespace, so they can be used in types, such as the type of length-indexed vectors:

```
data Vec :: Type -> Nat -> Type where
  Nil  :: Vec a Zero
  Cons :: a -> Vec a n -> Vec a (Succ n)
```

TypeData (page 363) is a more fine-grained alternative to the *DataKinds* (page 357) extension, which defines *all* the constructors in *all* `data` declarations as both data constructors and type constructors.

A `type data` declaration has the same syntax as a `data` declaration, either an ordinary algebraic data type or a GADT, prefixed with the keyword `type`, except that it may not contain a datatype context (even with *DatatypeContexts* (page 322)), labelled fields, strictness flags, or a `deriving` clause.

The only constraints permitted in the types of constructors are equality constraints, e.g.:

```
type data P :: Type -> Type -> Type where
  MkP :: (a ~ Natural, b ~~ Char) => P a b
```

Because `type data` declarations introduce type constructors, they do not permit constructors with the same names as types, so the following declaration is invalid:

```
type data T = T     // Invalid
```

The compiler will reject this declaration, because the type constructor `T` is defined twice (as the datatype being defined and as a type constructor).

The main type constructor of a `type data` declaration can be defined recursively, as in the `Nat` example above, but its constructors may not be used in types within the same mutually recursive group of declarations, so the following is forbidden:

```
type data T f = K (f (K Int))  // Invalid
```

6.4.13 Kind polymorphism

TypeInType

Implies *PolyKinds* (page 364), *DataKinds* (page 357), *KindSignatures* (page 511)

Since 8.0.1

Status Deprecated

The extension *TypeInType* (page 364) is now deprecated: its sole effect is to switch on *PolyKinds* (page 364) (and hence *KindSignatures* (page 511)) and *DataKinds* (page 357).

PolyKinds

Implies *KindSignatures* (page 511)

Since 7.4.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow kind polymorphic types.

This section describes GHC's kind system, as it appears in version 8.0 and beyond. The kind system as described here is always in effect, with or without extensions, although it is a conservative extension beyond standard Haskell. The extensions above simply enable syntax and tweak the inference algorithm to allow users to take advantage of the extra expressiveness of GHC's kind system.

6.4.13.1 Overview of kind polymorphism

Consider inferring the kind for

```
data App f a = MkApp (f a)
```

In Haskell 98, the inferred kind for `App` is `(Type -> Type) -> Type -> Type`. But this is overly specific, because another suitable Haskell 98 kind for `App` is `((Type -> Type) -> Type) -> (Type -> Type) -> Type`, where the kind assigned to `a` is `Type -> Type`. Indeed, without kind signatures (*KindSignatures* (page 511)), it is necessary to use a dummy constructor to get a Haskell compiler to infer the second kind. With kind polymorphism (*PolyKinds* (page 364)), GHC infers the kind `forall k. (k -> Type) -> k -> Type` for `App`, which is its most general kind.

Thus, the chief benefit of kind polymorphism is that we can now infer these most general kinds and use `App` at a variety of kinds:

```
App Maybe Int   -- `k` is instantiated to Type

data T a = MkT (a Int)    -- `a` is inferred to have kind (Type -> Type)
App T Maybe     -- `k` is instantiated to (Type -> Type)
```

6.4.13.2 Overview of Type-in-Type

GHC 8 extends the idea of kind polymorphism by declaring that types and kinds are indeed one and the same. Nothing within GHC distinguishes between types and kinds. Another way of thinking about this is that the type `Bool` and the "promoted kind" `Bool` are actually identical. (Note that term `True` and the type `'True` are still distinct, because the former can be used in expressions and the latter in types.) This lack of distinction between types and kinds is a hallmark of dependently typed languages. Full dependently typed languages also remove the difference between expressions and types, but doing that in GHC is a story for another day.

One simplification allowed by combining types and kinds is that the type of `Type` is just `Type`. It is true that the `Type :: Type` axiom can lead to non-termination, but this is not a problem in GHC, as we already have other means of non-terminating programs in both types and expressions. This decision (among many, many others) *does* mean that despite the expressiveness of GHC's type system, a "proof" you write in Haskell is not an irrefutable mathematical proof. GHC promises only partial correctness, that if your programs compile and run to completion, their results indeed have the types assigned. It makes no claim about programs that do not finish in a finite amount of time.

To learn more about this decision and the design of GHC under the hood please see the paper introducing this kind system to GHC/Haskell.

6.4.13.3 Principles of kind inference

Generally speaking, when *PolyKinds* (page 364) is on, GHC tries to infer the most general kind for a declaration. In many cases (for example, in a datatype declaration) the definition has a right-hand side to inform kind inference. But that is not always the case. Consider

```
type family F a
```

Type family declarations have no right-hand side, but GHC must still infer a kind for `F`. Since there are no constraints, it could infer `F :: forall k1 k2. k1 -> k2`, but that seems *too* polymorphic. So GHC defaults those entirely-unconstrained kind variables to `Type` and we get `F :: Type -> Type`. You can still declare `F` to be kind-polymorphic using kind signatures:

```
type family F1 a                -- F1 :: Type -> Type
type family F2 (a :: k)         -- F2 :: forall k. k -> Type
type family F3 a :: k           -- F3 :: forall k. Type -> k
type family F4 (a :: k1) :: k2  -- F4 :: forall k1 k2. k1 -> k2
```

The general principle is this:

- When there is a right-hand side, GHC infers the most polymorphic kind consistent with the right-hand side. Examples: ordinary data type and GADT declarations, class declarations. In the case of a class declaration the role of "right hand side" is played by the class method signatures.
- When there is no right hand side, GHC defaults argument and result kinds to `Type`, except when directed otherwise by a kind signature. Examples: data and open type family declarations.

This rule has occasionally-surprising consequences (see #10132).

```
class C a where    -- Class declarations are generalised
                   -- so C :: forall k. k -> Constraint
  data D1 a        -- No right hand side for these two family
```

(continues on next page)

(continued from previous page)

```
  type F1 a          -- declarations, but the class forces (a :: k)
                     -- so   D1, F1 :: forall k. k -> Type

data D2 a    -- No right-hand side so D2 :: Type -> Type
type F2 a    -- No right-hand side so F2 :: Type -> Type
```

The kind-polymorphism from the class declaration makes D1 kind-polymorphic, but not so D2; and similarly F1, F2.

6.4.13.4 Kind inference in type signatures

When kind-checking a type, GHC considers only what is written in that type when figuring out how to generalise the type's kind.

For example, consider these definitions (with *ScopedTypeVariables* (page 512)):

```
data Proxy a    -- Proxy :: forall k. k -> Type
p :: forall a. Proxy a
p = Proxy :: Proxy (a :: Type)
```

GHC reports an error, saying that the kind of a should be a kind variable k, not Type. This is because, by looking at the type signature `forall a. Proxy a`, GHC assumes a's kind should be generalised, not restricted to be Type. The function definition is then rejected for being more specific than its type signature.

6.4.13.5 Explicit kind quantification

Enabled by *PolyKinds* (page 364), GHC supports explicit kind quantification, as in these examples:

```
data Proxy :: forall k. k -> Type
f :: (forall k (a :: k). Proxy a -> ()) -> Int
```

Note that the second example has a `forall` that binds both a kind k and a type variable a of kind k. In general, there is no limit to how deeply nested this sort of dependency can work. However, the dependency must be well-scoped: `forall (a :: k) k. ...` is an error.

6.4.13.6 Inferring the order of variables in a type/class declaration

It is possible to get intricate dependencies among the type variables introduced in a type or class declaration. Here is an example:

```
data T a (b :: k) c = MkT (a c)
```

After analysing this declaration, GHC will discover that a and c can be kind-polymorphic, with `a :: k2 -> Type` and `c :: k2`. We thus infer the following kind:

```
T :: forall {k2 :: Type} (k :: Type). (k2 -> Type) -> k -> k2 -> Type
```

Note that k2 is placed *before* k, and that k is placed *before* a. Also, note that k2 is written here in braces. As explained with *TypeApplications* (page 386) (*Inferred vs. specified type variables* (page 387)), type and kind variables that GHC generalises over, but not written in the original program, are not available for visible type application. (These are called *inferred* variables.) Such variables are written in braces.

The general principle is this:

- Variables not available for type application come first.
- Then come variables the user has written, implicitly brought into scope in a type variable's kind.
- Lastly come the normal type variables of a declaration.
- Variables not given an explicit ordering by the user are sorted according to ScopedSort (*Ordering of specified variables* (page 388)).

With the T example above, we could bind k *after* a; doing so would not violate dependency concerns. However, it would violate our general principle, and so k comes first.

Sometimes, this ordering does not respect dependency. For example:

```
data T2 k (a :: k) (c :: Proxy '[a, b])
```

It must be that a and b have the same kind. Note also that b is implicitly declared in c's kind. Thus, according to our general principle, b must come *before* k. However, b *depends on* k. We thus reject T2 with a suitable error message.

In associated types, we order the type variables as if the type family was a top-level declaration, ignoring the visibilities of the class's type variable binders. Here is an example:

```
class C (a :: k) b where
  type F (c :: j) (d :: Proxy m) a b
```

We infer these kinds:

```
C :: forall {k1 :: Type} (k :: Type). k -> k1 -> Constraint
F :: forall {k1 :: Type} {k2 :: Type} {k3 :: Type} j (m :: k1).
     j -> Proxy m -> k2 -> k3 -> Type
```

Note that the kind of a is specified in the kind of C but inferred in the kind of F.

The "general principle" described here is meant to make all this more predictable for users. It would not be hard to extend GHC to relax this principle. If you should want a change here, consider writing a proposal to do so.

6.4.13.7 Complete user-supplied kind signatures and polymorphic recursion

CUSKs

Since 8.10.1

Status Included in *Haskell98* (page 272), *Haskell2010* (page 272)

NB! This is a legacy feature, see *StandaloneKindSignatures* (page 369) for the modern replacement.

Just as in type inference, kind inference for recursive types can only use *monomorphic* recursion. Consider this (contrived) example:

```
data T m a = MkT (m a) (T Maybe (m a))
-- GHC infers kind  T :: (Type -> Type) -> Type -> Type
```

The recursive use of T forced the second argument to have kind Type. However, just as in type inference, you can achieve polymorphic recursion by giving a *complete user-supplied kind signature* (or CUSK) for T. A CUSK is present when all argument kinds and the result kind are known, without any need for inference. For example:

```
data T (m :: k -> Type) :: k -> Type where
  MkT :: m a -> T Maybe (m a) -> T m a
```

The complete user-supplied kind signature specifies the polymorphic kind for T, and this signature is used for all the calls to T including the recursive ones. In particular, the recursive use of T is at kind Type.

What exactly is considered to be a "complete user-supplied kind signature" for a type constructor? These are the forms:

- For a datatype, every type variable must be annotated with a kind. In a GADT-style declaration, there may also be a kind signature (with a top-level `::` in the header), but the presence or absence of this annotation does not affect whether or not the declaration has a complete signature.

```
data T1 :: (k -> Type) -> k -> Type       where ...
-- Yes;  T1 :: forall k. (k->Type) -> k -> Type

data T2 (a :: k -> Type) :: k -> Type     where ...
-- Yes;  T2 :: forall k. (k->Type) -> k -> Type

data T3 (a :: k -> Type) (b :: k) :: Type where ...
-- Yes;  T3 :: forall k. (k->Type) -> k -> Type

data T4 (a :: k -> Type) (b :: k)      where ...
-- Yes;  T4 :: forall k. (k->Type) -> k -> Type

data T5 a (b :: k) :: Type             where ...
-- No;  kind is inferred

data T6 a b                         where ...
-- No;  kind is inferred
```

- For a datatype with a top-level `::`: all kind variables introduced after the `::` must be explicitly quantified.

```
data T1 :: k -> Type            -- No CUSK: `k` is not explicitly quantified
data T2 :: forall k. k -> Type  -- CUSK: `k` is bound explicitly
data T3 :: forall (k :: Type). k -> Type   -- still a CUSK
```

- For a newtype, the rules are the same as they are for a data type unless *UnliftedNewtypes* (page 558) is enabled. With *UnliftedNewtypes* (page 558), the type constructor only has a CUSK if a kind signature is present. As with a datatype with a top-level `::`, all kind variables introduced after the `::` must be explicitly quantified

```
{-# LANGUAGE UnliftedNewtypes #-}
newtype N1 where                 -- No; missing kind signature
newtype N2 :: TYPE IntRep where  -- Yes; kind signature present
newtype N3 (a :: Type) where     -- No; missing kind signature
newtype N4 :: k -> Type where    -- No; `k` is not explicitly quantified
newtype N5 :: forall (k :: Type). k -> Type where -- Yes; good signature
```

- For a class, every type variable must be annotated with a kind.

- For a type synonym, every type variable and the result type must all be annotated with kinds:

```
type S1 (a :: k) = (a :: k)    -- Yes   S1 :: forall k. k -> k
type S2 (a :: k) = a           -- No    kind is inferred
type S3 (a :: k) = Proxy a     -- No    kind is inferred
```

  Note that in S2 and S3, the kind of the right-hand side is rather apparent, but it is still not considered to have a complete signature – no inference can be done before detecting the signature.

- An un-associated open type or data family declaration *always* has a CUSK; un-annotated type variables default to kind Type:

```
data family D1 a                  -- D1 :: Type -> Type
data family D2 (a :: k)           -- D2 :: forall k. k -> Type
data family D3 (a :: k) :: Type   -- D3 :: forall k. k -> Type
type family S1 a :: k -> Type     -- S1 :: forall k. Type -> k -> Type
```

- An associated type or data family declaration has a CUSK precisely if its enclosing class has a CUSK.

```
class C a where                -- no CUSK
  type AT a b                  -- no CUSK, b is defaulted

class D (a :: k) where         -- yes CUSK
  type AT2 a b                 -- yes CUSK, b is defaulted
```

- A closed type family has a complete signature when all of its type variables are annotated and a return kind (with a top-level ::) is supplied.

It is possible to write a datatype that syntactically has a CUSK (according to the rules above) but actually requires some inference. As a very contrived example, consider

```
data Proxy a           -- Proxy :: forall k. k -> Type
data X (a :: Proxy k)
```

According to the rules above X has a CUSK. Yet, the kind of k is undetermined. It is thus quantified over, giving X the kind `forall k1 (k :: k1). Proxy k -> Type`.

The detection of CUSKs is enabled by the *CUSKs* (page 367) flag, which is switched off by default in GHC2021 and on in Haskell98 and Haskell2010. This extension is scheduled for deprecation to be replaced with *StandaloneKindSignatures* (page 369).

6.4.13.8 Standalone kind signatures and polymorphic recursion

StandaloneKindSignatures

Implies *NoCUSKs* (page 367)

Since 8.10.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Just as in type inference, kind inference for recursive types can only use *monomorphic* recursion. Consider this (contrived) example:

```
data T m a = MkT (m a) (T Maybe (m a))
-- GHC infers kind  T :: (Type -> Type) -> Type -> Type
```

The recursive use of T forced the second argument to have kind Type. However, just as in type inference, you can achieve polymorphic recursion by giving a *standalone kind signature* for T:

```
type T :: (k -> Type) -> k -> Type
data T m a = MkT (m a) (T Maybe (m a))
```

The standalone kind signature specifies the polymorphic kind for T, and this signature is used for all the calls to T including the recursive ones. In particular, the recursive use of T is at kind Type.

While a standalone kind signature determines the kind of a type constructor, it does not determine its arity. This is of particular importance for type families and type synonyms, as they cannot be partially applied. See *Type family declarations* (page 342) for more information about arity.

The arity can be specified using explicit binders and inline kind annotations:

```
-- arity F0 = 0
type F0 :: forall k. k -> Type
type family F0 :: forall k. k -> Type

-- arity F1 = 1
type F1 :: forall k. k -> Type
type family F1 :: k -> Type

-- arity F2 = 2
type F2 :: forall k. k -> Type
type family F2 a :: Type
```

In absence of an inline kind annotation, the inferred arity includes all explicitly bound parameters and all immediately following invisible parameters:

```
-- arity FD1 = 1
type FD1 :: forall k. k -> Type
type FD1

-- arity FD2 = 2
type FD2 :: forall k. k -> Type
type FD2 a
```

Note that F0, F1, F2, FD1, and FD2 all have identical standalone kind signatures. The arity is inferred from the type family header.

The kind variables bound by an outermost `forall` in a standalone kind signature scope only over the kind in that signature. Unlike term-level type signatures (see *Declaration type signatures* (page 513)), the outermost kind variables do *not* scope over the corresponding declaration. For example, given this class declaration:

```
class C (a :: k) where
  m :: Proxy k -> Proxy a -> String
```

The following would *not* be an equivalent definition of C:

```
type C :: forall k. k -> Constraint
class C a where
  m :: Proxy k -> Proxy a -> String
```

Because the k from the standalone kind signature does not scope over C's definition, the k in m's type signature is no longer the kind of a, but rather a completely distinct kind. It's as if you had written this:

```
type C :: forall k. k -> Constraint
class C (a :: kindOfA) where
  m :: forall k. Proxy k -> Proxy (a :: kindOfA) -> String
```

To avoid this issue, C's definition must be given an inline kind annotation like so:

```
type C :: forall k. k -> Constraint
class C (a :: k) where
  m :: Proxy k -> Proxy a -> String
```

6.4.13.9 Standalone kind signatures and declaration headers

GHC requires that in the presence of a standalone kind signature, data declarations must bind all their parameters For example:

```
type Prox1 :: k -> Type
data Prox1 a = MkProx1
  -- OK.

type Prox2 :: k -> Type
data Prox2 = MkProx2
  -- Error:
  --   • Expected a type, but found something with kind ‘k -> Type’
  --   • In the data type declaration for ‘Prox2’
```

GADT-style data declarations may either bind their parameters or use an inline signature in addition to the standalone kind signature:

```
type GProx1 :: k -> Type
data GProx1 a where MkGProx1 :: GProx1 a
  -- OK.

type GProx2 :: k -> Type
data GProx2 where MkGProx2 :: GProx2 a
  -- Error:
  --   • Expected a type, but found something with kind ‘k -> Type’
  --   • In the data type declaration for ‘GProx2’

type GProx3 :: k -> Type
data GProx3 :: k -> Type where MkGProx3 :: GProx3 a
  -- OK.

type GProx4 :: k1 -> Type
data GProx4 :: k2 -> Type where MkGProx4 :: GProx4 a
  -- OK.
```

Note that variables in a kind signature must stand for variables, not arbitrary types. For example, the following is rejected:

```
type GProx5 :: k -> Type
data GProx5 :: w where MkGProx5 :: GProx5 a
  -- Error:
```

(continues on next page)

(continued from previous page)

```
  --   • Couldn't match expected kind ‘w’ with actual kind ‘k -> Type’
  --   • In the data type declaration for ‘GProx5’
```

Classes are subject to the same rules:

```
type C1 :: Type -> Constraint
class C1 a
  -- OK.

type C2 :: Type -> Constraint
class C2
  -- Error:
  --   • Couldn't match expected kind ‘Constraint’
  --                 with actual kind ‘Type -> Constraint’
  --   • In the class declaration for ‘C2’
```

For type families, the number of parameters in the kind signature takes on additional meaning: it specifies the arity of the type family, i.e. how many arguments the type family requires before reducing. Any type family instances must then provide the same number of arguments. For example:

```
type F1 :: Type -> Type
type family F1 where
  F1 = Maybe
  -- OK.

type F2 :: Type -> Type
type family F2 where
  F2 () = Bool
  F2 a  = Maybe a
  -- Error:
  --   • Number of parameters must match family declaration; expected 0
  --   • In the type family declaration for `F2'

type F3 :: Type -> Type
type family F3 a where
  F3 () = Bool
  F3 a  = Maybe a
  -- OK.
```

Data families are tricky territory. Their headers are exempt from this rule, but their instances are not:

```
type T :: k -> Type
data family T
  -- OK.

data instance T Int = MkT1
  -- OK.

data instance T = MkT3
  -- Error:
  --   • Expecting one more argument to ‘T’
  --     Expected a type, but ‘T’ has kind ‘k0 -> Type’
  --   • In the data instance declaration for ‘T’
```

This also applies to GADT-style data instances:

```
data instance T (a :: Nat) where MkN4 :: T 4
                                 MKN9 :: T 9
  -- OK.

data instance T :: Symbol -> Type where MkSN :: T "Neptune"
                                        MkSJ :: T "Jupiter"
  -- OK.

data instance T where MkT4 :: T x
  -- Error:
  --    • Expecting one more argument to 'T'
  --      Expected a type, but 'T' has kind 'k0 -> Type'
  --    • In the data instance declaration for 'T'
```

6.4.13.10 Kind inference in data type declarations

Consider the declaration

```
data T1 f a = MkT1 (f a)
data T2 f a where
  MkT2 :: f a -> T f a
```

In both cases GHC looks at the data constructor declarations to give constraints on the kind of `T`, yielding

```
T1, T2 :: forall k. (k -> Type) -> k -> Type
```

Consider the type

```
type G :: forall k. k -> Type
data G (a :: k) where
  GInt    :: G Int
  GMaybe  :: G Maybe
```

This datatype `G` is GADT-like in both its kind and its type. Suppose you have `g :: G a`, where `a :: k`. Then pattern matching to discover that `g` is in fact `GMaybe` tells you both that `k ~ (Type -> Type)` and `a ~ Maybe`. The definition for `G` requires that *`PolyKinds`* (page 364) be in effect, but pattern-matching on `G` requires no extension beyond *`GADTs`* (page 335). That this works is actually a straightforward extension of regular GADTs and a consequence of the fact that kinds and types are the same.

Note that the datatype `G` is used at different kinds in its body, and therefore that kind-indexed GADTs use a form of polymorphic recursion. It is thus only possible to use this feature if you have provided a complete user-supplied kind signature (CUSK) for the datatype (*Complete user-supplied kind signatures and polymorphic recursion* (page 367)), or a standalone kind signature (*Standalone kind signatures and polymorphic recursion* (page 369)); in the case of `G` we both. If you wish to see the kind indexing explicitly, you can do so by enabling *`-fprint-explicit-kinds`* (page 77) and querying `G` with GHCi's *`:info`* (page 48) command:

```
> :set -fprint-explicit-kinds
> :info G
type role G nominal nominal
type G :: forall k. k -> Type
data G @k a where
```

(continues on next page)

(continued from previous page)

```
  GInt   :: G @Type Int
  GMaybe :: G @(Type -> Type) Maybe
```

where you can see the GADT-like nature of the two constructors.

6.4.13.11 Kind inference for data/newtype instance declarations

Consider these declarations

```
data family T :: forall k. (k -> Type) -> k -> Type

data instance T p q where
   MkT :: forall r. r Int -> T r Int
```

Here T has an invisible kind argument; and perhaps it is instantiated to Type in the instance, thus:

```
data instance T @Type (p :: Type -> Type) (q :: Type) where
   MkT :: forall r. r Int -> T r Int
```

Or perhaps we intended the specialisation to be in the GADT data constructor, thus:

```
data instance T @k (p :: k -> Type) (q :: k) where
   MkT :: forall r. r Int -> T @Type r Int
```

It gets more complicated if there are multiple constructors. In general, there is no principled way to tell which type specialisation comes from the data instance, and which from the individual GADT data constructors.

So GHC implements this rule: in data/newtype instance declararations (unlike ordinary data/newtype declarations) we do *not* look at the constructor declarations when inferring the shape of the instance header. The principle is that *the instantiation of the data instance should be apparent from the header alone*. This principle makes the program easier to understand, and avoids the swamp of complexity indicated above.

6.4.13.12 Kind inference in class instance declarations

Consider the following example of a poly-kinded class and an instance for it:

```
class C a where
  type F a

instance C b where
  type F b = b -> b
```

In the class declaration, nothing constrains the kind of the type a, so it becomes a poly-kinded type variable (a :: k). Yet, in the instance declaration, the right-hand side of the associated type instance b -> b says that b must be of kind Type. GHC could theoretically propagate this information back into the instance head, and make that instance declaration apply only to type of kind Type, as opposed to types of any kind. However, GHC does *not* do this.

In short: GHC does *not* propagate kind information from the members of a class instance declaration into the instance declaration head.

This lack of kind inference is simply an engineering problem within GHC, but getting it to work would make a substantial change to the inference infrastructure, and it's not clear the payoff is worth it. If you want to restrict b's kind in the instance above, just use a kind signature in the instance head.

6.4.13.13 Kind inference in type synonyms and type family instances

Consider the scoping rules for type synonyms and type family instances, such as these:

```
type          TS a (b :: k) = <rhs>
type instance TF a (b :: k) = <rhs>
```

The basic principle is that all variables mentioned on the right hand side <rhs> must be bound on the left hand side:

```
type TS a (b :: k) = (k, a, Proxy b)    -- accepted
type TS a (b :: k) = (k, a, Proxy b, z) -- rejected: z not in scope
```

But there is one exception: free variables mentioned in the outermost kind signature on the right hand side are quantified implicitly. Thus, in the following example the variables a, b, and k are all in scope on the right hand side of S:

```
type S a b = <rhs> :: k -> k
```

The reason for this exception is that there may be no other way to bind k. For example, suppose we wanted S to have the following kind with an *invisible* parameter k:

```
S :: forall k. Type -> Type -> k -> k
```

In this case, we could not simply bind k on the left-hand side, as k would become a *visible* parameter:

```
type S k a b = <rhs> :: k -> k
S :: forall k -> Type -> Type -> k -> k
```

Note that we only look at the *outermost* kind signature to decide which variables to quantify implicitly. As a counter-example, consider M1:

```
type M1 = Just (Nothing :: Maybe k)    -- rejected: k not in scope
```

Here, the kind signature is hidden inside Just, and there is no outermost kind signature. We can fix this example by providing an outermost kind signature:

```
type M2 = Just (Nothing :: Maybe k) :: Maybe (Maybe k)
```

Here, k is brought into scope by `:: Maybe (Maybe k)`.

A kind signature is considered to be outermost regardless of redundant parentheses:

```
type P =    Nothing :: Maybe a    -- accepted
type P = (((Nothing :: Maybe a))) -- accepted
```

Closed type family instances are subject to the same rules:

```
type family F where
  F = Nothing :: Maybe k           -- accepted

type family F where
  F = Just (Nothing :: Maybe k)     -- rejected: k not in scope

type family F where
  F = Just (Nothing :: Maybe k) :: Maybe (Maybe k)  -- accepted

type family F :: Maybe (Maybe k) where
  F = Just (Nothing :: Maybe k)     -- rejected: k not in scope

type F :: forall k. Maybe (Maybe k)
type family F @k where
  F @k = Just (Nothing :: Maybe k)  -- accepted

-- CUSKs version (Legacy)
type family F :: Maybe (Maybe k) where
  F @k = Just (Nothing :: Maybe k)  -- accepted
```

Kind variables can also be quantified in *visible* positions. Consider the following two examples:

```
data ProxyKInvis (a :: k)
data ProxyKVis k (a :: k)
```

In the first example, the kind variable `k` is an *invisible* argument to `ProxyKInvis`. In other words, a user does not need to instantiate `k` explicitly, as kind inference automatically determines what `k` should be. For instance, in `ProxyKInvis True`, `k` is inferred to be `Bool`. This is reflected in the kind of `ProxyKInvis`:

```
ProxyKInvis :: forall k. k -> Type
```

In the second example, `k` is a *visible* argument to `ProxyKVis`. That is to say, `k` is an argument that users must provide explicitly when applying `ProxyKVis`. For example, `ProxyKVis Bool True` is a well formed type.

What is the kind of `ProxyKVis`? One might say `forall k. Type -> k -> Type`, but this isn't quite right, since this would allow incorrect things like `ProxyKVis Bool Int`, which should be rejected due to the fact that `Int` is not of kind `Bool`. The key observation is that the kind of the second argument *depends* on the first argument. GHC indicates this dependency in the syntax that it gives for the kind of `ProxyKVis`:

```
ProxyKVis :: forall k -> k -> Type
```

This kind is similar to the kind of `ProxyKInvis`, but with a key difference: the type variables quantified by the `forall` are followed by an arrow (`->`), not a dot (`.`). This is a visible, dependent quantifier. It is visible in that the user must pass in a type for `k` explicitly, and it is dependent in the sense that `k` appears later in the kind of `ProxyKVis`. As a counterpart, the `k` binder in `forall k. k -> Type` can be thought of as an *invisible*, dependent quantifier.

GHC permits writing kinds with this syntax, provided that the `ExplicitForAll` and `PolyKinds` language extensions are enabled. Just like the invisible `forall`, one can put explicit kind signatures on visibly bound kind variables, so the following is syntactically valid:

```
ProxyKVis :: forall (k :: Type) -> k -> Type
```

Currently, the ability to write visible, dependent quantifiers is limited to kinds. Consequently, visible dependent quantifiers are rejected in any context that is unambiguously the type of a term. They are also rejected in the types of data constructors.

6.4.13.14 Kind inference in closed type families

Although all open type families are considered to have a complete user-supplied kind signature (*Complete user-supplied kind signatures and polymorphic recursion* (page 367)), we can relax this condition for closed type families, where we have equations on which to perform kind inference. GHC will infer kinds for the arguments and result types of a closed type family.

GHC supports *kind-indexed* type families, where the family matches both on the kind and type. GHC will *not* infer this behaviour without a complete user-supplied kind signature or standalone kind signature (see *Standalone kind signatures and polymorphic recursion* (page 369)), because doing so would sometimes infer non-principal types. Indeed, we can see kind-indexing as a form of polymorphic recursion, where a type is used at a kind other than its most general in its own definition.

For example:

```
type family F1 a where
  F1 True  = False
  F1 False = True
  F1 x     = x
-- F1 fails to compile: kind-indexing is not inferred

type family F2 (a :: k) where
  F2 True  = False
  F2 False = True
  F2 x     = x
-- F2 fails to compile: no complete signature

type F3 :: k -> k
type family F3 a where
  F3 True  = False
  F3 False = True
  F3 x     = x

-- CUSKs version (legacy)
type family F4 (a :: k) :: k where
  F4 True  = False
  F4 False = True
  F4 x     = x
-- OK
```

6.4.13.15 Higher-rank kinds

In concert with *RankNTypes* (page 402), GHC supports higher-rank kinds. Here is an example:

```
-- Heterogeneous propositional equality
data (a :: k1) :~~: (b :: k2) where
  HRefl :: a :~~: a

class HTestEquality (t :: forall k. k -> Type) where
  hTestEquality :: forall k1 k2 (a :: k1) (b :: k2). t a -> t b -> Maybe (a :~~: b)
```

Note that `hTestEquality` takes two arguments where the type variable `t` is applied to types of different kinds. That type variable must then be polykinded. Accordingly, the kind of `HTestEquality` (the class) is `(forall k. k -> Type) -> Constraint`, a higher-rank kind.

A big difference with higher-rank kinds as compared with higher-rank types is that `forall`s in kinds *cannot* be moved. This is best illustrated by example. Suppose we want to have an instance of `HTestEquality` for `(:~~:)`.

```
instance HTestEquality ((:~~:) a) where
  hTestEquality HRefl HRefl = Just HRefl
```

With the declaration of `(:~~:)` above, it gets kind `forall k1 k2. k1 -> k2 -> Type`. Thus, the type `(:~~:) a` has kind `k2 -> Type` for some `k2`. GHC cannot then *regeneralize* this kind to become `forall k2. k2 -> Type` as desired. Thus, the instance is rejected as ill-kinded.

To allow for such an instance, we would have to define `(:~~:)` as follows:

```
data (:~~:) :: forall k1. k1 -> forall k2. k2 -> Type where
  HRefl :: a :~~: a
```

In this redefinition, we give an explicit kind for `(:~~:)`, deferring the choice of `k2` until after the first argument (`a`) has been given. With this declaration for `(:~~:)`, the instance for `HTestEquality` is accepted.

6.4.13.16 The kind `Type`

StarIsType

Since 8.6.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271), *Haskell2010* (page 272), *Haskell98* (page 272)

Treat the unqualified uses of the `*` type operator as nullary and desugar to `Data.Kind.Type`.

The kind `Type` (imported from `Data.Kind`) classifies ordinary types. With *StarIsType* (page 378) (currently enabled by default), `*` is desugared to `Type`, but using this legacy syntax is not recommended due to conflicts with *TypeOperators* (page 323). This also applies to `★`, the Unicode variant of `*`.

6.4.13.17 Inferring dependency in datatype declarations

If a type variable a in a datatype, class, or type family declaration depends on another such variable k in the same declaration, two properties must hold:

- a must appear after k in the declaration, and
- k must appear explicitly in the kind of *some* type variable in that declaration.

The first bullet simply means that the dependency must be well-scoped. The second bullet concerns GHC's ability to infer dependency. Inferring this dependency is difficult, and GHC currently requires the dependency to be made explicit, meaning that k must appear in the kind of a type variable, making it obvious to GHC that dependency is intended. For example:

```
data Proxy k (a :: k)            -- OK: dependency is "obvious"
data Proxy2 k a = P (Proxy k a)  -- ERROR: dependency is unclear
```

In the second declaration, GHC cannot immediately tell that k should be a dependent variable, and so the declaration is rejected.

It is conceivable that this restriction will be relaxed in the future, but it is (at the time of writing) unclear if the difficulties around this scenario are theoretical (inferring this dependency would mean our type system does not have principal types) or merely practical (inferring this dependency is hard, given GHC's implementation). So, GHC takes the easy way out and requires a little help from the user.

6.4.13.18 Inferring dependency in user-written `forall`s

A programmer may use `forall` in a type to introduce new quantified type variables. These variables may depend on each other, even in the same `forall`. However, GHC requires that the dependency be inferrable from the body of the `forall`. Here are some examples:

```
data Proxy k (a :: k) = MkProxy   -- just to use below

f :: forall k a. Proxy k a        -- This is just fine. We see that (a :: k).
f = undefined

g :: Proxy k a -> ()              -- This is to use below.
g = undefined

data Sing a
h :: forall k a. Sing k -> Sing a -> ()  -- No obvious relationship between k and a
h _ _ = g (MkProxy :: Proxy k a)  -- This fails. We didn't know that a should have␣
↪kind k.
```

Note that in the last example, it's impossible to learn that a depends on k in the body of the `forall` (that is, the `Sing k -> Sing a -> ()`). And so GHC rejects the program.

6.4.13.19 Kind defaulting without PolyKinds

Without *PolyKinds* (page 364), GHC refuses to generalise over kind variables. It thus defaults kind variables to Type when possible; when this is not possible, an error is issued.

Here is an example of this in action:

```
{-# LANGUAGE PolyKinds #-}
import Data.Kind (Type)
data Proxy a = P   -- inferred kind: Proxy :: k -> Type
data Compose f g x = MkCompose (f (g x))
  -- inferred kind: Compose :: (b -> Type) -> (a -> b) -> a -> Type

-- separate module having imported the first
{-# LANGUAGE NoPolyKinds, DataKinds #-}
z = Proxy :: Proxy MkCompose
```

In the last line, we use the promoted constructor MkCompose, which has kind

```
forall (a :: Type) (b :: Type) (f :: b -> Type) (g :: a -> b) (x :: a).
  f (g x) -> Compose f g x
```

Now we must infer a type for z. To do so without generalising over kind variables, we must default the kind variables of MkCompose. We can easily default a and b to Type, but f and g would be ill-kinded if defaulted. The definition for z is thus an error.

6.4.13.20 Pretty-printing in the presence of kind polymorphism

With kind polymorphism, there is quite a bit going on behind the scenes that may be invisible to a Haskell programmer. GHC supports several flags that control how types are printed in error messages and at the GHCi prompt. See the *discussion of type pretty-printing options* (page 76) for further details. If you are using kind polymorphism and are confused as to why GHC is rejecting (or accepting) your program, we encourage you to turn on these flags, especially *-fprint-explicit-kinds* (page 77).

6.4.13.21 Datatype return kinds

With *KindSignatures* (page 511), we can give the kind of a datatype written in GADT-syntax (see *GADTSyntax* (page 329)). For example:

```
data T :: Type -> Type where ...
```

There are a number of restrictions around these *return kinds*. The text below considers *UnliftedNewtypes* (page 558) and data families (enabled by *TypeFamilies* (page 338)). The discussion also assumes familiarity with *representation polymorphism* (page 381).

1. data and data instance declarations must have return kinds that end in TYPE LiftedRep. (Recall that Type is just a synonym for TYPE LiftedRep.) By "end in", we refer to the kind left over after all arguments (introduced either by forall or ->) are stripped off and type synonyms expanded. Note that no type family expansion is done when performing this check.
2. If *UnliftedNewtypes* (page 558) is enabled, then newtype and newtype instance declarations must have return kinds that end in TYPE rep for some rep. The rep may mention type families, but the TYPE must be apparent without type family expansion. (Type synonym expansion is acceptable.)

If *UnliftedNewtypes* (page 558) is not enabled, then `newtype` and `newtype instance` declarations have the same restrictions as `data` declarations.

3. A `data` or `newtype` instance actually can have *two* return kinds. The first is the kind derived by applying the data family to the patterns provided in the instance declaration. The second is given by a kind annotation. Both return kinds must satisfy the restrictions above.

Examples:

```
data T1 :: Type              -- good: Type expands to TYPE LiftedRep
data T2 :: TYPE LiftedRep    -- good
data T3 :: forall k. k -> Type -> Type  -- good: arguments are dropped

type LR = LiftedRep
data T3 :: TYPE LR           -- good: we look through type synonyms

type family F a where
  F Int = LiftedRep

data T4 :: TYPE (F Int)      -- bad: we do not look through type families

type family G a where
  G Int = Type

data T5 :: G Int             -- bad: we do not look through type families

-- assume -XUnliftedNewtypes
newtype T6 :: Type where ...              -- good
newtype T7 :: TYPE (F Int) where ...      -- good
newtype T8 :: G Int where ...             -- bad

data family DF a :: Type
data instance DF Int :: Type              -- good
data instance DF Bool :: TYPE LiftedRep   -- good
data instance DF Char :: G Int            -- bad

data family DF2 k :: k                    -- good
data family DF2 Type                      -- good
data family DF2 Bool                      -- bad
data family DF2 (G Int)                   -- bad for 2 reasons:
                                          --  a type family can't be in a pattern, and
                                          --  the kind fails the restrictions here
```

6.4.14 Representation polymorphism

In order to allow full flexibility in how kinds are used, it is necessary to use the kind system to differentiate between boxed, lifted types (normal, everyday types like `Int` and `[Bool]`) and unboxed, primitive types (*Unboxed types and primitive operations* (page 554)) like `Int#`. We thus have so-called representation polymorphism.

Here are the key definitions, all available from `GHC.Exts`:

```
TYPE :: RuntimeRep -> Type   -- highly magical, built into GHC

data Levity = Lifted    -- for things like `Int`
            | Unlifted  -- for things like `Array#`
```

(continues on next page)

(continued from previous page)

```
data RuntimeRep = BoxedRep Levity  -- for anything represented by a GC-managed pointer
                | IntRep           -- for `Int#`
                | TupleRep [RuntimeRep]  -- unboxed tuples, indexed by the␣
↪representations of the elements
                | SumRep [RuntimeRep]    -- unboxed sums, indexed by the␣
↪representations of the disjuncts
                | ...

type LiftedRep = BoxedRep Lifted

type Type = TYPE LiftedRep    -- Type is just an ordinary type synonym
```

The idea is that we have a new fundamental type constant `TYPE`, which is parameterised by a `RuntimeRep`. We thus get `Int# :: TYPE IntRep` and `Bool :: TYPE LiftedRep`. Anything with a type of the form `TYPE x` can appear to either side of a function arrow `->`. We can thus say that `->` has type `TYPE r1 -> TYPE r2 -> TYPE LiftedRep`. The result is always lifted because all functions are lifted in GHC.

6.4.14.1 Levity polymorphism

A special case of representation polymorphism is levity polymorphism, where we abstract over a variable of kind `Levity`, such as:

```
example :: forall (l :: Levity) (a :: TYPE (BoxedRep l)). (Int -> a) -> a
example f = f 42
```

With *UnliftedDatatypes* (page 559), we can even declare levity-polymorphic data types:

```
type PEither :: Type -> Type -> TYPE (BoxedRep l)
data PEither l r = PLeft l | PRight r
```

6.4.14.2 No representation-polymorphic variables or arguments

If GHC didn't have to compile programs that run in the real world, that would be the end of the story. But representation polymorphism can cause quite a bit of trouble for GHC's code generator. Consider

```
bad :: forall (r1 :: RuntimeRep) (r2 :: RuntimeRep)
              (a :: TYPE r1) (b :: TYPE r2).
       (a -> b) -> a -> b
bad f x = f x
```

This seems like a generalisation of the standard `$` operator. If we think about compiling this to runnable code, though, problems appear. In particular, when we call `bad`, we must somehow pass `x` into `bad`. How wide (that is, how many bits) is `x`? Is it a pointer? What kind of register (floating-point or integral) should `x` go in? It's all impossible to say, because `x`'s type, `a :: TYPE r1` is representation-polymorphic. We thus forbid such constructions, via the following straightforward rule:

> No variable may have a representation-polymorphic type.

This eliminates `bad` because the variable `x` would have a representation-polymorphic type.

However, not all is lost. We can still do this:

```
($) :: forall r (a :: Type) (b :: TYPE r).
       (a -> b) -> a -> b
f $ x = f x
```

Here, only `b` is representation-polymorphic. There are no variables with a representation-polymorphic type. And the code generator has no trouble with this. Indeed, this is the true type of GHC's `$` operator, slightly more general than the Haskell 98 version.

Because the code generator must store and move arguments as well as variables, the logic above applies equally well to function arguments, which may not be representation-polymorphic.

6.4.14.3 Representation-polymorphic bottoms

We can use representation polymorphism to good effect with `error` and `undefined`, whose types are given here:

```
undefined :: forall (r :: RuntimeRep) (a :: TYPE r).
             HasCallStack => a
error :: forall (r :: RuntimeRep) (a :: TYPE r).
         HasCallStack => String -> a
```

These functions do not bind a representation-polymorphic variable, and so are accepted. Their polymorphism allows users to use these to conveniently stub out functions that return unboxed types.

6.4.14.4 Inference and defaulting

GHC does not infer representation-polymorphic types. If the representation of a variable is not specified, it will be assumed to be `LiftedRep`. For example, if you write `f a b = a b`, the inferred type of `f` will be

```
f :: forall {a :: Type} {b :: Type}. (a -> b) -> a -> b
```

even though

```
f :: forall {rep} {a :: Type} {b :: TYPE rep}. (a -> b) -> a -> b
```

would also be legal, as described above.

Likewise, in a user-written signature `f :: forall a b. (a -> b) -> a -> b` GHC will assume that both `a` and `b` have kind `Type`. To use a different representation, you have to specify the kinds of `a` and `b`.

During type inference, GHC does not quantify over variables of kind `RuntimeRep` nor `Levity`. Instead, they are defaulted to `LiftedRep` and `Lifted` respectively. Likewise, `Multiplicity` variables (*Linear types* (page 408)) are defaulted to `Many`.

6.4.14.5 Printing representation-polymorphic types

-fprint-explicit-runtime-reps

Print `RuntimeRep` and `Levity` parameters as they appear; otherwise, they are defaulted to `LiftedRep` and `Lifted`, respectively.

Most GHC users will not need to worry about representation polymorphism or unboxed types. For these users, seeing the representation polymorphism in the type of `$` is unhelpful. And thus, by default, it is suppressed, by supposing all type variables of type `RuntimeRep` to be `LiftedRep` when printing, and printing `TYPE LiftedRep` as `Type` (or `*` when *StarIsType* (page 378) is on).

Should you wish to see representation polymorphism in your types, enable the flag *-fprint-explicit-runtime-reps* (page 384). For example,

```
ghci> :t ($)
($) :: (a -> b) -> a -> b
ghci> :set -fprint-explicit-runtime-reps
ghci> :t ($)
($)
  :: forall (r :: GHC.Types.RuntimeRep) a (b :: TYPE r).
     (a -> b) -> a -> b
```

6.4.15 Type-Level Literals

GHC supports numeric, string, and character literals at the type level, giving convenient access to a large number of predefined type-level constants. Numeric literals are of kind `Natural`, string literals are of kind `Symbol`, and character literals are of kind `Char`. This feature is enabled by the *DataKinds* (page 357) language extension.

The kinds of the literals and all other low-level operations for this feature are defined in modules `GHC.TypeLits` and `GHC.TypeNats`. Note that these modules define some type-level operators that clash with their value-level counterparts (e.g. `(+)`). Import and export declarations referring to these operators require an explicit namespace annotation (see *Explicit namespaces in import/export* (page 319)).

Here is an example of using type-level numeric literals to provide a safe interface to a low-level function:

```haskell
import GHC.TypeLits
import Data.Word
import Foreign

newtype ArrPtr (n :: Natural) a = ArrPtr (Ptr a)

clearPage :: ArrPtr 4096 Word8 -> IO ()
clearPage (ArrPtr p) = ...
```

Also type-level naturals could be promoted from the `Natural` data type using *DataKinds*, for example:

```haskell
data Point = MkPoint Natural Natural
type MyCoordinates = MkPoint 95 101
```

Here is an example of using type-level string literals to simulate simple record operations:

```
data Label (l :: Symbol) = Get

class Has a l b | a l -> b where
  from :: a -> Label l -> b

data Point = Point Int Int deriving Show

instance Has Point "x" Int where from (Point x _) _ = x
instance Has Point "y" Int where from (Point _ y) _ = y

example = from (Point 1 2) (Get :: Label "x")
```

6.4.15.1 Runtime Values for Type-Level Literals

Sometimes it is useful to access the value-level literal associated with a type-level literal. This is done with the functions `natVal` and `symbolVal`. For example:

```
GHC.TypeLits> natVal (Proxy :: Proxy 2)
2
```

These functions are overloaded because they need to return a different result, depending on the type at which they are instantiated.

```
natVal :: KnownNat n => proxy n -> Natural  -- from GHC.TypeNats
natVal :: KnownNat n => proxy n -> Integer  -- from GHC.TypeLits

-- instance KnownNat 0
-- instance KnownNat 1
-- instance KnownNat 2
-- ...
```

GHC discharges the constraint as soon as it knows what concrete type-level literal is being used in the program. Note that this works only for *literals* and not arbitrary type expressions. For example, a constraint of the form `KnownNat (a + b)` will *not* be simplified to `(KnownNat a, KnownNat b)`; instead, GHC will keep the constraint as is, until it can simplify `a + b` to a constant value.

It is also possible to convert a run-time integer or string value to the corresponding type-level literal. Of course, the resulting type literal will be unknown at compile-time, so it is hidden in an existential type. The conversion may be performed using `someNatVal` for integers and `someSymbolVal` for strings:

```
someNatVal :: Natural -> Maybe SomeNat  -- from GHC.TypeNats
someNatVal :: Integer -> Maybe SomeNat  -- from GHC.TypeLits

SomeNat    :: KnownNat n => Proxy n -> SomeNat
```

The operations on strings are similar.

6.4.15.2 Computing With Type-Level Naturals

GHC 7.8 can evaluate arithmetic expressions involving type-level natural numbers. Such expressions may be constructed using the type-families `(+)`, `(*)`, `(^)` for addition, multiplication, and exponentiation. Numbers may be compared using `(<=?)`, which returns a promoted boolean value, or `(<=)`, which compares numbers as a constraint. For example:

```
GHC.TypeLits> natVal (Proxy :: Proxy (2 + 3))
5
```

At present, GHC is quite limited in its reasoning about arithmetic: it will only evaluate the arithmetic type functions and compare the results— in the same way that it does for any other type function. In particular, it does not know more general facts about arithmetic, such as the commutativity and associativity of `(+)`, for example.

However, it is possible to perform a bit of "backwards" evaluation. For example, here is how we could get GHC to compute arbitrary logarithms at the type level:

```
lg :: Proxy base -> Proxy (base ^ pow) -> Proxy pow
lg _ _ = Proxy

GHC.TypeLits> natVal (lg (Proxy :: Proxy 2) (Proxy :: Proxy 8))
3
```

6.4.16 Visible type application

TypeApplications

Since 8.0.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow the use of type application syntax.

The *TypeApplications* (page 386) extension allows you to use *visible type application* in expressions. Here is an example: `show (read @Int "5")`. The `@Int` is the visible type application; it specifies the value of the type variable in `read`'s type.

A visible type application is preceded with an `@` sign. (To disambiguate the syntax, the `@` must be preceded with a non-identifier letter, usually a space. For example, `read@Int 5` would not parse.) It can be used whenever the full polymorphic type of the function is known. If the function is an identifier (the common case), its type is considered known only when the identifier has been given a type signature. If the identifier does not have a type signature, visible type application cannot be used.

GHC also permits visible kind application, where users can declare the kind arguments to be instantiated in kind-polymorphic cases. Its usage parallels visible type application in the term level, as specified above.

In addition to visible type application in terms and types, the type application syntax can be used in patterns matching a data constructor to bind type variables in that constructor's type.

6.4.16.1 Inferred vs. specified type variables

GHC tracks a distinction between what we call *inferred* and *specified* type variables. Only specified type variables are available for instantiation with visible type application. An example illustrates this well:

```
f :: (Eq b, Eq a) => a -> b -> Bool
f x y = (x == x) && (y == y)

g x y = (x == x) && (y == y)
```

The functions `f` and `g` have the same body, but only `f` is given a type signature. When GHC is figuring out how to process a visible type application, it must know what variable to instantiate. It thus must be able to provide an ordering to the type variables in a function's type.

If the user has supplied a type signature, as in `f`, then this is easy: we just take the ordering from the type signature, going left to right and using the first occurrence of a variable to choose its position within the ordering. Thus, the variables in `f` will be `b`, then `a`.

In contrast, there is no reliable way to do this for `g`; we will not know whether `Eq a` or `Eq b` will be listed first in the constraint in `g`'s type. In order to have visible type application be robust between releases of GHC, we thus forbid its use with `g`.

We say that the type variables in `f` are *specified*, while those in `g` are *inferred*. The general rule is this: if the user has written a type variable in the source program, it is *specified*; if not, it is *inferred*.

This rule applies in datatype declarations, too. For example, if we have `data Proxy a = Proxy` (and *PolyKinds* (page 364) is enabled), then `a` will be assigned kind `k`, where `k` is a fresh kind variable. Because `k` was not written by the user, it will be unavailable for type application in the type of the constructor `Proxy`; only the `a` will be available.

Inferred variables are printed in braces. Thus, the type of the data constructor `Proxy` from the previous example is `forall {k} (a :: k). Proxy a`. We can observe this behavior in a GHCi session:

```
> :set -XTypeApplications -fprint-explicit-foralls
> let myLength1 :: Foldable f => f a -> Int; myLength1 = length
> :type myLength1
myLength1 :: forall (f :: * -> *) a. Foldable f => f a -> Int
> let myLength2 = length
> :type myLength2
myLength2 :: forall {t :: * -> *} {a}. Foldable t => t a -> Int
> :type myLength2 @[]

<interactive>:1:1: error:
    • Cannot apply expression of type ‘t0 a0 -> Int’
      to a visible type argument ‘[]’
    • In the expression: myLength2 @[]
```

Notice that since `myLength1` was defined with an explicit type signature, *:type* (page 54) reports that all of its type variables are available for type application. On the other hand, `myLength2` was not given a type signature. As a result, all of its type variables are surrounded with braces, and trying to use visible type application with `myLength2` fails.

6.4.16.2 Ordering of specified variables

In the simple case of the previous section, we can say that specified variables appear in left-to-right order. However, not all cases are so simple. Here are the rules in the subtler cases:

- If an identifier's type has a `forall`, then the order of type variables as written in the `forall` is retained.
- If any of the variables depend on other variables (that is, if some of the variables are *kind* variables), the variables are reordered so that kind variables come before type variables, preserving the left-to-right order as much as possible. That is, GHC performs a stable topological sort on the variables. Example:

  ```
  h :: Proxy (a :: (j, k)) -> Proxy (b :: Proxy a) -> ()
    -- as if h :: forall j k a b. ...
  ```

  In this example, `a` depends on `j` and `k`, and `b` depends on `a`. Even though `a` appears lexically before `j` and `k`, `j` and `k` are quantified first, because `a` depends on `j` and `k`. Note further that `j` and `k` are not reordered with respect to each other, even though doing so would not violate dependency conditions.

  A "stable topological sort" here, we mean that we perform this algorithm (which we call *ScopedSort*):

  - Work left-to-right through the input list of type variables, with a cursor.
  - If variable `v` at the cursor is depended on by any earlier variable `w`, move `v` immediately before the leftmost such `w`.
- Class methods' type arguments include the class type variables, followed by any variables an individual method is polymorphic in. So, `class Monad m where return :: a -> m a` means that `return`'s type arguments are `m, a`.
- With the *`RankNTypes`* (page 402) extension (*Lexically scoped type variables* (page 512)), it is possible to declare type arguments somewhere other than the beginning of a type. For example, we can have `pair :: forall a. a -> forall b. b -> (a, b)` and then say `pair @Bool True @Char` which would have type `Char -> (Bool, Char)`.
- Partial type signatures (*Partial Type Signatures* (page 520)) work nicely with visible type application. If you want to specify only the second type argument to `wurble`, then you can say `wurble @_ @Int`. The first argument is a wildcard, just like in a partial type signature. However, if used in a visible type application/visible kind application, it is *not* necessary to specify *`PartialTypeSignatures`* (page 520) and your code will not generate a warning informing you of the omitted type.

The section in this manual on kind polymorphism describes how variables in type and class declarations are ordered (*Inferring the order of variables in a type/class declaration* (page 366)).

6.4.16.3 Manually defining inferred variables

Since the 9.0.1 release, GHC permits labelling the user-written type or kind variables as *inferred*, in contrast to the default of *specified*. By writing the type variable binder in braces as `{tyvar}` or `{tyvar :: kind}`, the new variable will be classified as inferred, not specified. Doing so gives the programmer control over which variables can be manually instantiated and which can't. Note that the braces do not influence scoping: variables in braces are still brought into scope just the same. Consider for example:

```
myConst :: forall {a} b. a -> b -> a
myConst x _ = x
```

In this example, despite both variables appearing in a type signature, `a` is an inferred variable while `b` is specified. This means that the expression `myConst @Int` has type `forall {a}. a -> Int -> a`.

The braces are allowed in the following places:

- In the type signatures of functions, variables, class methods, as well as type annotations on expressions. Consider the example above.
- In data constructor declarations, using the GADT syntax. Consider:

  ```
  data T a where MkT :: forall {k} (a :: k). Proxy a -> T a
  ```

  The constructor `MkT` defined in this example is kind polymorphic, which is emphasized to the reader by explicitly abstracting over the `k` variable. As this variable is marked as inferred, it can not be manually instantiated.
- In existential variable quantifications, e.g.:

  ```
  data HList = HNil
             | forall {a}. HCons a HList
  ```
- In pattern synonym signatures. Consider for instance:

  ```
  data T a where MkT :: forall a b. a -> b -> T a

  pattern Pat :: forall {c}. () => forall {d}. c -> d -> T c
  pattern Pat x y = MkT x y
  ```

  Note that in this example, `a` is a universal variable in the data type `T`, where `b` is existential. When writing the pattern synonym, both types are allowed to be specified or inferred.
- On the right-hand side of a type synonym, e.g.:

  ```
  type Foo = forall a {b}. Either a b
  ```
- In type signatures on variables bound in RULES, e.g.:

  ```
  {-# RULES "parametricity" forall (f :: forall {a}. a -> a). map f = id #-}
  ```

The braces are *not* allowed in the following places:

- In visible dependent quantifiers. Consider:

  ```
  data T :: forall {k} -> k -> Type
  ```

This example is rejected, as a visible argument should by definition be explicitly applied. Making them inferred (and thus not appliable) would be conflicting.

- In SPECIALISE pragmas or in instance declaration heads, e.g.:

```
instance forall {a}. Eq (Maybe a) where ...
```

The reason for this is, essentially, that none of these define a new construct. This means that no new type is being defined where specificity could play a role.

- On the left-hand sides of type declarations, such as classes, data types, etc.

Note that while specified and inferred type variables have different properties vis-à-vis visible type application, they do not otherwise affect GHC's notion of equality over types. For example, given the following definitions:

```
id1 :: forall a. a -> a
id1 x = x

id2 :: forall {a}. a -> a
id2 x = x

app1 :: (forall a. a -> a) -> b -> b
app1 g x = g x

app2 :: (forall {a}. a -> a) -> b -> b
app2 g x = g x
```

GHC will deem all of `app1 id1`, `app1 id2`, `app2 id1`, and `app2 id2` to be well typed.

6.4.17 Type abstractions

TypeAbstractions

Since 9.8.1

Status Experimental

Allow the use of type abstraction syntax.

The *TypeAbstractions* (page 390) extension provides a way to explicitly bind scoped type or kind variables using the `@a` syntax. The feature is only partially implemented, and this text covers only the implemented parts, whereas the full specification can be found in GHC Proposals #448 and #425.

6.4.17.1 Type Abstractions in Patterns

Since: GHC 9.2

The type abstraction syntax can be used in patterns that match a data constructor. The syntax can't be used with record patterns or infix patterns. This is useful in particular to bind existential type variables associated with a GADT data constructor as in the following example:

```
{-# LANGUAGE AllowAmbiguousTypes #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE TypeApplications #-}
```

(continues on next page)

(continued from previous page)

```
import Data.Proxy

data Foo where
  Foo :: forall a. (Show a, Num a) => Foo

test :: Foo -> String
test x = case x of
  Foo @t -> show @t 0

main :: IO ()
main = print $ test (Foo @Float)
```

In this example, the case in `test` is binding an existential variable introduced by `Foo` that otherwise could not be named and used.

It's possible to bind variables to any part of the type arguments to a constructor; there is no need for them to be existential. In addition, it's possible to "match" against part of the type argument using type constructors.

For a somewhat-contrived example:

```
foo :: (Num a) => Maybe [a] -> String
foo (Nothing @[t]) = show (0 :: t)
foo (Just @[t] xs) = show (sum xs :: t)
```

Here, we're binding the type variable t to be the type of the elements of the list type which is itself the argument to Maybe.

The order of the type arguments specified by the type applications in a pattern is the same as that for an expression: either the order as given by the user in an explicit `forall` in the definition of the data constructor, or if that is not present, the order in which the type variables appear in its type signature from left to right.

For example if we have the following declaration in GADT syntax:

```
data Foo :: * -> * where
  A :: forall s t. [(t,s)] -> Foo (t,s)
  B :: (t,s) -> Foo (t,s)
```

Then the type arguments to `A` will match first `s` and then `t`, while the type arguments to `B` will match first `t` and then `s`.

Type arguments appearing in patterns can influence the inferred type of a definition:

```
foo (Nothing @Int) = 0
foo (Just x) = x
```

will have inferred type:

```
foo :: Maybe Int -> Int
```

which is more restricted than what it would be without the application:

```
foo :: Num a => Maybe a -> a
```

For more information and detail regarding type applications in patterns, see the paper Type variables in patterns by Eisenberg, Breitner and Peyton Jones. Relative to that paper, the implementation in GHC for now at least makes one additional conservative restriction, that type variables occurring in patterns must not already be in scope, and so are always new variables that only bind whatever type is matched, rather than ever referring to a variable from an outer scope. Type wildcards _ may be used in any place where no new variable needs binding.

6.4.17.2 Type Abstractions in Functions

Since: GHC 9.10

Type abstraction syntax can be used in lambdas and function left-hand sides to bring into scope type variables associated with invisible `forall`. For example:

```
id :: forall a. a -> a
id @t x = x :: t
```

Here type variables `t` and `a` stand for the same type, i.e. the first and only type argument of `id`. In a call `id @Int` we have `a = Int`, `t = Int`. The difference is that `a` is in scope in the type signature, while `t` is in scope in the function equation.

The scope of `a` can be extended to cover the function equation as well by enabling *ScopedTypeVariables* (page 512). Using a separate binder like `@t` is the modern and more flexible alternative for that, capable of handling higher-rank scenarios (see the `higherRank` example below).

When multiple variables are bound with `@`-binders, they are matched left-to-right with the corresponding forall-bound variables in the type signature:

```
const :: forall a. forall b. a -> b -> a
const @ta @tb x  = x
```

In this example, `@ta` corresponds to `forall a.` and `@tb` to `forall b.`. It is also possible to use `@`-binders in combination with implicit quantification (i.e. no explicit forall in the signature):

```
const :: a -> b -> a
const @ta @tb x  = x
```

In such cases, type variables in the signature are considered to be quantified with an implicit `forall` in the order in which they appear in the signature, c.f. *TypeApplications* (page 386).

It is not possible to match against a specific type (such as `Maybe` or `Int`) in an `@`-binder. The binder must be irrefutable, i.e. it may take one of the following forms:

- type variable pattern `@a`
- type variable pattern with a kind annotation `@(f :: Type -> Type)`
- wildcard `@_`, with or without a kind annotation

The main advantage to using `@`-binders over *ScopedTypeVariables* (page 512) is the ability to use them in lambdas passed to higher-rank functions:

```
higherRank :: (forall a. (Num a, Bounded a) => a -> a) -> (Int8, Int16)
higherRank f = (f 42, f 42)

ex :: (Int8, Int16)
ex = higherRank (\ @a x -> maxBound @a - x )
                   -- @a-binder in a lambda pattern in an argument
                   -- to a higher-order function
```

At the moment, an @-binder is valid only in a limited set of circumstances:

- In a function left-hand side, where the function must have an explicit type signature:

```
f1 :: forall a. a -> forall b. b -> (a, b)
f1 @a x @b y = (x :: a, y :: b)        -- OK
```

  It would be illegal to omit the type signature for `f`, nor is it possible to move the binder to a lambda on the RHS:

```
f2 :: forall a. a -> forall b. b -> (a, b)
f2 = \ @a x @b y -> (x :: a, y :: b)   -- ILLEGAL
```

- In a lambda annotated with an inline type signature:

```
f3 = (\ @a x @b y -> (x :: a, y :: b) )      -- OK
    :: forall a. a -> forall b. b -> (a, b)
```

- In a lambda used as an argument to a higher-rank function or data constructor:

```
h :: (forall a. a -> forall b. b -> (a, b)) -> (Int, Bool)
h = ...

f4 = h (\ @a x @b y -> (x :: a, y :: b))     -- OK
```

- In a lambda used as a field of a data structure (e.g. a list item), whose type is impredicative (see *ImpredicativeTypes* (page 407)):

```
f5 :: [forall a. a -> a -> a]
f5 = [ \ @a x _ -> x :: a,
       \ @a _ y -> y :: a ]
```

- In a lambda of multiple arguments, where the first argument is visible, and only if *DeepSubsumption* (page 405) is off:

```
{-# LANGUAGE NoDeepSubsumption #-}
f6 :: () -> forall a. a -> (a, a)
f6 = \ _ @a x -> (x :: a, x)   -- OK
```

6.4.17.3 Invisible Binders in Type Declarations

Since: GHC 9.8

Syntax

The type abstraction syntax can be used in type declaration headers, including `type`, `data`, `newtype`, `class`, `type family`, and `data family` declarations. Here are a few examples:

```
type C :: forall k. k -> Constraint
class C @k a where ...
        ^^

type D :: forall k j. k -> j -> Type
data D @k @j (a :: k) (b :: j) = ...
       ^^ ^^
```

(continues on next page)

(continued from previous page)

```
type F :: forall p q. p -> q -> (p, q)
type family F @p @q a b where ...
              ^^ ^^
```

Just as ordinary type parameters, invisible type variable binders may have kind annotations:

```
type F :: forall p q. p -> q -> (p, q)
type family F @(p :: Type) @(q :: Type) (a :: p) (b :: q) where ...
```

Scope

The `@k`-binders scope over the body of the declaration and can be used to bring implicit type or kind variables into scope. Consider:

```
type C :: forall i. (i -> i -> i) -> Constraint
class C @i a where
    p :: P a i
```

Without the `@i` binder in `C @i a`, the `i` in `P a i` would no longer refer to the class variable `i` and would be implicitly quantified in the method signature instead.

Type checking

Invisible type variable binders require either a standalone kind signature or a complete user-supplied kind.

If a standalone kind signature is given, GHC will match up `@k`-binders with the corresponding `forall k.` quantifiers in the signature:

```
type B :: forall k. k -> forall j. j -> Type
data B @k (a :: k) @j (b :: j)
```

Quantifier-binder pairs of B	
`forall k.`	`@k`
`k ->`	`(a :: k)`
`forall j.`	`@j`
`j ->`	`(b :: j)`

The matching is done left-to-right. Consider:

```
type S :: forall a b. a -> b -> Type
type S @k x y = ...
```

In this example, `@k` is matched with `forall a.`, not `forall b.`:

Quantifier-binder pairs of S	
`forall a.`	`@k`
`forall b.`	
`a ->`	`x`
`b ->`	`y`

When a standalone kind signature is absent but the definition has a complete user-supplied kind (and the *CUSKs* (page 367) extension is enabled), a @k-binder gives rise to a `forall k.` quantifier in the inferred kind signature. The inferred `forall k.` does not float to the left; the order of quantifiers continues to match the order of binders in the header:

```
-- Inferred kind: forall k. k -> forall j. j -> Type
data B @(k :: Type) (a :: k) @(j :: Type) (b :: j)
```

6.4.18 Required type arguments

RequiredTypeArguments

Since 9.10.1

Status Experimental

Allow visible dependent quantification `forall x ->` in types of terms.

This feature is only partially implemented in GHC. In this section we describe the implemented subset, while the full specification can be found in GHC Proposal #281.

The *RequiredTypeArguments* (page 395) extension enables the use of visible dependent quantification in types of terms:

```
id     :: forall a.   a -> a         -- invisible dependent quantification
id_vdq :: forall a -> a -> a         --   visible dependent quantification
```

The arrow in `forall a ->` is part of the syntax and not a function arrow, just like the dot in `forall a.` is not a type operator.

The choice between `forall a.` and `forall a ->` does not have any effect on program execution. Both quantifiers introduce type variables, which are erased during compilation. Rather, the main difference is in the syntax used at call sites:

```
x1 = id       True     -- invisible forall, the type argument is inferred by GHC
x2 = id @Bool True     -- invisible forall, the type argument is supplied by the␣
↪programmer

x3 = id_vdq _    True  --   visible forall, the type argument is inferred by GHC
x4 = id_vdq Bool True  --   visible forall, the type argument is supplied by the␣
↪programmer
```

6.4.18.1 Terminology: Dependent quantifier

Both `forall a.` and `forall a ->` are said to be "dependent" because the result type depends on the supplied type argument:

```
id @Integer :: Integer -> Integer
id @String  :: String  -> String

id_vdq Integer :: Integer -> Integer
id_vdq String  :: String  -> String
```

Notice how the RHS of the signature is influenced by the LHS.

This is in contrast to the function arrow `->`, which is a non-dependent quantifier:

```
putStrLn "Hello" :: IO ()
putStrLn "World" :: IO ()
```

The type of `putStrLn` is `String -> IO ()`. No matter what string we pass as input, the result type `IO ()` does not depend on it.

This notion of dependence is weaker than the one used in dependently-typed languages (see *Relation to Π-types* (page 401)).

6.4.18.2 Terminology: Visible quantifier

We say that `forall a.` is an *invisible* quantifier and `forall a ->` is a *visible* quantifier. This notion of "visibility" is unrelated to implicit quantification, which happens when the quantifier is omitted:

```
id     ::             a -> a    -- implicit quantification, invisible forall
id     :: forall a.   a -> a    -- explicit quantification, invisible forall
id_vdq :: forall a -> a -> a    -- explicit quantification,   visible forall
```

The property of "visibility" actually describes whether the corresponding type argument is visible at the definition site and at call sites:

```
-- Invisible quantification
id :: forall a. a -> a
id x = x                -- defn site: `a` is not mentioned
call_id = id True       -- call site: `a` is invisibly instantiated to `Bool`

-- Visible quantification
id_vdq :: forall a -> a -> a
id_vdq t x = x                  -- defn site: `a` is visibly bound to `t`
call_id_vdq = id_vdq Bool True  -- call site: `a` is visibly instantiated to `Bool`
```

In the equation for `id` there is just one binder on the LHS, `x`, and it corresponds to the value argument, not to the type argument. Compare that with the definition of `id_vdq`:

```
id_vdq :: forall a -> a -> a
id_vdq t x = x
```

This time we have two binders on the LHS:

- `t`, corresponding to `forall a ->` in the signature
- `x`, corresponding to `a ->` in the signature

The bound `t` can be used in subsequent patterns, as well as on the right-hand side of the equation:

```
id_vdq :: forall a -> a -> a
id_vdq t (x :: t) = x :: t
--     ↑       ↑         ↑
--   bound    used      used
```

We use the terms "visible type argument" and "required type argument" interchangeably.

6.4.18.3 Relation to TypeApplications

RequiredTypeArguments (page 395) are similar to *TypeApplications* (page 386) in that we pass a type to a function as an explicit argument. The difference is that type applications are optional: it is up to the caller whether to write `id @Bool True` or `id True`. By default, the compiler infers that the type variable is instantiated to `Bool`. The existence of a type argument is not reflected syntactically in the expression, it is invisible unless we use a *visibility override*, i.e. `@`.

Required type arguments are compulsory. They must appear syntactically at call sites:

```
x1 = id_vdq Bool True    -- OK
x2 = id_vdq      True    -- not OK
```

You may use an underscore to infer a required type argument:

```
x3 = id_vdq _ True       -- OK
```

That is, it is mostly a matter of syntax whether to use `forall a.` with type applications or `forall a ->`. One advantage of required type arguments is that they are never ambiguous. Consider the type of `Foreign.Storable.sizeOf`:

```
sizeOf :: forall a. Storable a => a -> Int
```

The value parameter is not actually used, its only purpose is to drive type inference. At call sites, one might write `sizeOf (undefined :: Bool)` or `sizeOf @Bool undefined`. Either way, the `undefined` is entirely superfluous and exists only to avoid an ambiguous type variable.

With *RequiredTypeArguments* (page 395), we can imagine a slightly different API:

```
sizeOf :: forall a -> Storable a => Int
```

If `sizeOf` had this type, we could write `sizeOf Bool` without passing a dummy value.

Required type arguments are erased during compilation. While the source program appears to bind and pass required type arguments alongside value arguments, the compiled program does not. There is no runtime overhead associated with required type arguments relative to the usual, invisible type arguments.

6.4.18.4 Relation to ExplicitNamespaces

A required type argument is syntactically indistinguishable from a value argument. In a function call `f arg1 arg2 arg3`, it is impossible to tell, without looking at the type of `f`, which of the three arguments are required type arguments, if any.

At the same time, one of the design goals of GHC is to be able to perform name resolution (find the binding sites of identifiers) without involving the type system. Consider:

```
data Ty = Int | Double | String deriving Show
main = print Int
```

In this example, there are two constructors named `Int` in scope:

- The **type constructor** `Int` of kind `Type` (imported from `Prelude`)
- The **data constructor** `Int` of type `Ty` (defined locally)

How does the compiler or someone reading the code know that `print Int` is supposed to refer to the data constructor, not the type constructor? In GHC, this is resolved as follows. Each identifier is said to occur either in **type syntax** or **term syntax**, depending on the surrounding syntactic context:

```
-- Examples of X in type syntax
type T = X      -- RHS of a type synonym
data D = MkD X  -- field of a data constructor declaration
a :: X          -- RHS of a type signature
b = f (c :: X)  -- RHS of a type signature (in expressions)
f (x :: X) = x  -- RHS of a type signature (in patterns)

-- Examples of X in term syntax
c X = a         -- LHS of a function equation
c a = X         -- RHS of a function equation
```

One could imagine the entire program "zoned" into type syntax and term syntax, each zone having its own rules for name resolution:

- In type syntax, type constructors take precedence over data constructors.
- In term syntax, data constructors take precedence over type constructors.

This means that in the `print Int` example, the data constructor is selected solely based on the fact that the `Int` occurs in term syntax. This is firmly determined before GHC attempts to type-check the expression, so the type of `print` does not influence which of the two `Int`s is passed to it.

This may not be the desired behavior in a required type argument. Consider:

```
vshow :: forall a -> Show a => a -> String
vshow t x = show (x :: t)

s1 = vshow Int    42      -- "42"
s2 = vshow Double 42      -- "42.0"
```

The function calls `vshow Int 42` and `vshow Double 42` are written in *term* syntax, while the intended referents of `Int` and `Double` are the respective *type* constructors. As long as there are no data constructors named `Int` or `Double` in scope, the example works as intended. However, if such clashing constructor names are introduced, they may disrupt name resolution:

```
data Ty = Int | Double | String

vshow :: forall a -> Show a => a -> String
vshow t x = show (x :: t)

s1 = vshow Int    42      -- error: Expected a type, but 'Int' has kind 'Ty'
s2 = vshow Double 42      -- error: Expected a type, but 'Double' has kind 'Ty'
```

In this example the intent was to refer to `Int` and `Double` as types, but the names were resolved in favor of data constructors, resulting in type errors.

The example can be fixed with the help of *`ExplicitNamespaces`* (page 319), which allows embedding type syntax into term syntax using the `type` keyword:

```
s1 = vshow (type Int)    42
s2 = vshow (type Double) 42
```

A similar problem occurs with list and tuple syntax. In type syntax, `[a]` is the type of a list, i.e. `Data.List.List a`. In term syntax, `[a]` is a singleton list, i.e. `a : []`. A naive attempt to use the list type as a required type argument will result in a type error:

```
s3 = vshow [Int] [1,2,3]  -- error: Expected a type, but '[Int]' has kind '[Type]'
```

The problem is that GHC assumes `[Int]` to stand for `Int : []` instead of the intended `Data.List.List Int`. This, too, can be solved using the `type` keyword:

```
s3 = vshow (type [Int]) [1,2,3]
```

Since the `type` keyword is merely a namespace disambiguation mechanism, it need not apply to the entire type argument. Using it to disambiguate only a part of the type argument is also valid:

```
f :: forall a -> ...   -- `f`` is a function that expects a required type argument

r1 = f (type (Either () Int))           -- `type` applied to the entire type argument
r2 = f (Either (type ()) Int)           -- `type` applied to one part of it
r3 = f (Either (type ()) (type Int))    -- `type` applied to multiple parts
```

That is, the expression `Either (type ()) (type Int)` does *not* indicate that `Either` is applied to two type arguments; rather, the entire expression is a single type argument and `type` is used to disambiguate parts of it.

Outside a required type argument, it is illegal to use `type`:

```
r4 = type Int  -- illegal use of 'type'
```

Finally, there are types that require the `type` keyword only due to limitations of the current implementation:

```
a1 = f (type (Int -> Bool))                          -- function type
a2 = f (type (Read T => T))                          -- constrained type
a3 = f (type (forall a. a))                          -- universally quantified type
a4 = f (type (forall a. Read a => String -> a))      -- a combination of the above
```

This restriction will be relaxed in a future release of GHC.

6.4.18.5 Effect on implicit quantification

Implicit quantification is said to occur when GHC inserts an implicit `forall` to bind type variables:

```
const :: a -> b -> a                -- implicit quantification
const :: forall a b. a -> b -> a    -- explicit quantification
```

Normally, implicit quantification is unaffected by term variables in scope:

```
f a = ...  -- the LHS binds `a`
  where const :: a -> b -> a
          -- implicit quantification over `a` takes place
          -- despite the `a` bound on the LHS of `f`
```

When *RequiredTypeArguments* (page 395) is in effect, names bound in term syntax are not implicitly quantified. This allows us to accept the following example:

```
readshow :: forall a -> (Read a, Show a) => String -> String
readshow t s = show (read s :: t)

s1 = readshow Int    "42"      -- "42"
s2 = readshow Double "42"      -- "42.0"
```

Note how `t` is bound on the LHS of a function equation (term syntax), and then used in a type annotation (type syntax). Under the usual rules for implicit quantification, the `t` would have been implicitly quantified:

```
-- RequiredTypeArguments
readshow t s = show (read s :: t)   -- the `t` is captured
--       ↑                     ↑
--      bound                 used

-- NoRequiredTypeArguments
readshow t s = show (read s :: t)   -- the `t` is implicitly quantified as follows:
readshow t s = show (read s :: forall t. t)
--       ↑                            ↑  ↑
--      bound                     bound used
```

On the one hand, taking the current scope into account allows us to accept programs like the one above. On the other hand, some existing programs will no longer compile:

```
a = 42
f :: a -> a    -- RequiredTypeArguments: the top-level `a` is captured
```

Because of that, merely enabling *RequiredTypeArguments* (page 395) might lead to type errors of this form:

```
Term variable ‘a’ cannot be used here
  (term variables cannot be promoted)
```

There are two possible ways to fix this error:

```
a = 42
f1 :: b -> b              -- (1) use a different variable name
f2 :: forall a. a -> a    -- (2) use an explicit forall
```

If you are converting a large codebase to be compatible with *RequiredTypeArguments* (page 395), consider using *-Wterm-variable-capture* (page 109) during the migration. It is a warning that detects instances of implicit quantification incompatible with *RequiredTypeArguments* (page 395):

```
The type variable ‘a’ is implicitly quantified,
even though another variable of the same name is in scope:
  ‘a’ defined at ...
```

6.4.18.6 Relation to Π-types

Both `forall a.` and `forall a ->` are dependent quantifiers in the narrow sense defined in *Terminology: Dependent quantifier* (page 395). However, neither of them constitutes a dependent function type (Π-type) that might be familiar to users coming from dependently-typed languages or proof assistants.

- Haskell has always had functions whose result *value* depends on the argument *value*:

  ```
  not True  = False   -- argument value: True;  result value: False
  (*2) 5    = 10      -- argument value: 5;     result value: 10
  ```

  This captures the usual idea of a function, denoted `a -> b`.

- Haskell also has functions whose result *type* depends on the argument *type*:

  ```
  id    @Int  :: Int  -> Int    -- argument type: Int;  result type: Int  -> Int
  id_vdq Bool :: Bool -> Bool   -- argument type: Bool; result type: Bool -> Bool
  ```

  This captures the idea of parametric polymorphism, denoted `forall a. b` or `forall a -> b`.

- Furthermore, Haskell has functions whose result *value* depends on the argument *type*:

  ```
  maxBound @Int8   = 127    -- argument type: Int8;  result value: 127
  maxBound @Int16  = 32767  -- argument type: Int16; result value: 32767
  ```

  This captures the idea of ad-hoc (class-based) polymorphism, denoted `C a => b`.

- However, Haskell does **not** have direct support for functions whose result *type* depends on the argument *value*. In the literature, these are often called "dependent functions", or "Π-types".

  Consider:

  ```
  type F :: Bool -> Bool
  type family F b where
    F True  = ...
    F False = ...

  f :: Bool -> Bool
  f True  = ...
  f False = ...
  ```

  In this example, we define a type family `F` to pattern-match on `b` at the type level; and a function `f` to pattern-match on `b` at the term level. However, it is impossible to quantify over `b` in such a way that both `F` and `f` could be applied to it:

  ```
  depfun :: forall (b :: Bool) -> F b  -- Allowed
  depfun b = ... (f b) ...             -- Not allowed
  ```

  It is illegal to pass `b` to `f` because `b` does not exist at runtime. Types and type arguments are erased before runtime.

The *`RequiredTypeArguments`* (page 395) extension does not add dependent functions, which would be a much bigger step. Rather *`RequiredTypeArguments`* (page 395) just makes it possible for the type arguments of a function to be compulsory.

6.4.19 Arbitrary-rank polymorphism

RankNTypes

> **Implies** *ExplicitForAll* (page 506)
>
> **Since** 6.8.1
>
> **Status** Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow types of arbitrary rank.

Rank2Types

> **Since** 6.8.1
>
> **Status** Deprecated

A deprecated alias of *RankNTypes* (page 402).

GHC's type system supports *arbitrary-rank* explicit universal quantification in types. For example, all the following types are legal:

```
f1 :: forall a b. a -> b -> a
g1 :: forall a b. (Ord a, Eq  b) => a -> b -> a

f2 :: (forall a. a->a) -> Int -> Int
g2 :: (forall a. Eq a => [a] -> a -> Bool) -> Int -> Int

f3 :: ((forall a. a->a) -> Int) -> Bool -> Bool
```

Here, `f1` and `g1` are rank-1 types, and can be written in standard Haskell (e.g. `f1 :: a->b->a`). The `forall` makes explicit the universal quantification that is implicitly added by Haskell.

The functions `f2` and `g2` have rank-2 types; the `forall` is on the left of a function arrow. As `g2` shows, the polymorphic type on the left of the function arrow can be overloaded.

The function `f3` has a rank-3 type; it has rank-2 types on the left of a function arrow.

The language option *RankNTypes* (page 402) (which implies *ExplicitForAll* (page 506)) enables higher-rank types. That is, you can nest `forall`s arbitrarily deep in function arrows. For example, a forall-type (also called a "type scheme"), including a type-class context, is legal:

- On the left or right of a function arrow.
- As the argument of a constructor, or type of a field, in a data type declaration. For example, any of the `f1, f2, f3, g1, g2` above would be valid field type signatures.
- As the type of an implicit parameter.
- In a pattern type signature (see *Lexically scoped type variables* (page 512)).

In particular, in `data` and `newtype` declarations the constructor arguments may be polymorphic types of any rank; see examples in *Examples* (page 403). Note that the declared types are nevertheless always monomorphic. This is important because by default GHC will not instantiate type variables to a polymorphic type (*Impredicative polymorphism* (page 407)).

Note that the *RankNTypes* (page 402) option is also required for any type with a `forall` or context to the right of an arrow. For example:

```
h1  :: Int -> (forall a. a -> a)
h1' :: forall a. Int -> (a -> a)

k1  :: Int -> Ord a => a -> a
k1' :: Ord a => Int -> a -> a
```

The function `h1` has a rank-1 type; it has the same behaviour as `h1'`, except with a different order of arguments. This matters if one were to specify the type explicitly using a visible type application (using *TypeApplications* (page 386)): we would write `h1 3 @Bool True` but `h1' @Bool 3 True`. Similarly, `k1` has a rank-1 type; it differs from `k1'` only in the order of arguments. As the types of `h1` and `k1` are not allowed in Haskell-98, we also require users to enable *RankNTypes* (page 402) to write them (which seems more sensible than inventing a separate extension just for this case).

The obsolete language option *Rank2Types* (page 402) is a synonym for *RankNTypes* (page 402). They used to specify finer distinctions that GHC no longer makes.

6.4.19.1 Examples

These are examples of `data` and `newtype` declarations whose data constructors have polymorphic argument types:

```
data T a = T1 (forall b. b -> b -> b) a

data MonadT m = MkMonad { return :: forall a. a -> m a,
                          bind   :: forall a b. m a -> (a -> m b) -> m b
                        }

newtype Swizzle = MkSwizzle (forall a. Ord a => [a] -> [a])
```

The constructors have rank-2 types:

```
T1 :: forall a. (forall b. b -> b -> b) -> a -> T a

MkMonad :: forall m. (forall a. a -> m a)
                  -> (forall a b. m a -> (a -> m b) -> m b)
                  -> MonadT m

MkSwizzle :: (forall a. Ord a => [a] -> [a]) -> Swizzle
```

In earlier versions of GHC, it was possible to omit the `forall` in the type of the constructor if there was an explicit context. For example:

```
newtype Swizzle' = MkSwizzle' (Ord a => [a] -> [a])
```

Since GHC 8.0 declarations such as `MkSwizzle'` will cause an out-of-scope error.

You construct values of types `T1, MonadT, Swizzle` by applying the constructor to suitable values, just as usual. For example,

```
a1 :: T Int
a1 = T1 (\x y->x) 3

a2, a3 :: Swizzle
a2 = MkSwizzle sort
a3 = MkSwizzle reverse
```

(continues on next page)

(continued from previous page)

```
a4 :: MonadT Maybe
a4 = let r x = Just x
     b m k = case m of
           Just y -> k y
           Nothing -> Nothing
     in
     MkMonad r b

mkTs :: (forall b. b -> b -> b) -> a -> a -> [T a]
mkTs f x y = [T1 f x, T1 f y]
```

The type of the argument can, as usual, be more general than the type required, as (`MkSwizzle reverse`) shows. (`reverse` does not need the `Ord` constraint.)

When you use pattern matching, the bound variables may now have polymorphic types. For example:

```
f :: T a -> a -> (a, Char)
f (T1 w k) x = (w k x, w 'c' 'd')

g :: (Ord a, Ord b) => Swizzle -> [a] -> (a -> b) -> [b]
g (MkSwizzle s) xs f = s (map f (s xs))

h :: MonadT m -> [m a] -> m [a]
h m [] = return m []
h m (x:xs) = bind m x          $ \y ->
             bind m (h m xs)   $ \ys ->
             return m (y:ys)
```

In the function `h` we use the record selectors `return` and `bind` to extract the polymorphic bind and return functions from the `MonadT` data structure, rather than using pattern matching.

6.4.19.2 Subsumption

Suppose:

```
f1 :: (forall a b. Int -> a -> b -> b) -> Bool
g1 :: forall x y. Int -> y -> x -> x

f2 :: (forall a. (Eq a, Show a) => a -> a) -> Bool
g2 :: forall x. (Show x, Eq x) => x -> x
```

then `f1 g1` and `f2 g2` are both well typed, despite the different order of type variables and constraints. What happens is that the argument is instantiated, and then re-generalised to match the type expected by the function.

But this instantiation and re-generalisation happens only at the top level of a type. In particular, none of this happens if the foralls are underneath an arrow. For example:

```
f3 :: (Int -> forall a b. a -> b -> b) -> Bool
g3a :: Int -> forall x y. x -> y -> y
g3b :: forall x. Int -> forall y. x -> y -> y
g3c :: Int -> forall x y. y -> x -> x
```

(continues on next page)

(continued from previous page)

```
f4 :: (Int -> forall a. (Eq a, Show a) => a -> a) -> Bool
g4 ::  Int -> forall x. (Show x, Eq x) => x -> x) -> Bool
```

Then the application `f3 g3a` is well-typed, because `g3a` has a type that matches the type expected by `f3`. But `f3 g3b` is not well typed, because the foralls are in different places. Nor is `f3 g3c`, where the foralls are in the same place but the variables are in a different order. Similarly `f4 g4` is not well typed, because the constraints appear in a different order.

These examples can be made to typecheck by eta-expansion. For example `f3 (\x -> g3b x)` is well typed, and similarly `f3 (\x -> g3c x)` and `f4 (\x -> g4 x)`.

A similar phenomenon occurs for operator sections. For example, ``(\`g3a\` "hello")`` is not well typed, but it can be made to typecheck by eta expanding it to ``\x -> x \`g3a\` "hello"``.

DeepSubsumption

Since 9.2.4

Relax the simple subsumption rules, implicitly inserting eta-expansions when matching up function types with different quantification structures.

The *DeepSubsumption* (page 405) extension relaxes the aforementioned requirement that foralls must appear in the same place. GHC will instead automatically rewrite expressions like `f x` of type `ty1 -> ty2` to become `(\ (y :: ty1) -> f x y)`; this is called eta-expansion. See Section 4.6 of Practical type inference for arbitrary-rank types, where this process is called "deep skolemisation".

Note that these eta-expansions may silently change the semantics of the user's program:

```
h1 :: Int -> forall a. a -> a
h1 = undefined
h2 :: forall b. Int -> b -> b
h2 = h1
```

With *DeepSubsumption* (page 405), GHC will accept these definitions, inserting an implicit eta-expansion:

```
h2 = \ i -> h1 i
```

This means that ``h2 `seq` ()`` will not crash, even though ``h1 `seq` ()`` does crash.

Historical note: Deep skolemisation was initially removed from the language by GHC Proposal #287, but was re-introduced as part of the *DeepSubsumption* (page 405) extension following GHC Proposal #511.

6.4.19.3 Type inference

In general, type inference for arbitrary-rank types is undecidable. GHC uses an algorithm proposed by Odersky and Laufer ("Putting type annotations to work", POPL'96) to get a decidable algorithm by requiring some help from the programmer. We do not yet have a formal specification of "some help" but the rule is this:

> For a lambda-bound or case-bound variable, x, either the programmer provides an explicit polymorphic type for x, or GHC's type inference will assume that x's type has no foralls in it.

What does it mean to "provide" an explicit type for x? You can do that by giving a type signature for x directly, using a pattern type signature (*Lexically scoped type variables* (page 512)), thus:

```
\ f :: (forall a. a->a) -> (f True, f 'c')
```

Alternatively, you can give a type signature to the enclosing context, which GHC can "push down" to find the type for the variable:

```
(\ f -> (f True, f 'c')) :: (forall a. a->a) -> (Bool,Char)
```

Here the type signature on the expression can be pushed inwards to give a type signature for f. Similarly, and more commonly, one can give a type signature for the function itself:

```
h :: (forall a. a->a) -> (Bool,Char)
h f = (f True, f 'c')
```

You don't need to give a type signature if the lambda bound variable is a constructor argument. Here is an example we saw earlier:

```
f :: T a -> a -> (a, Char)
f (T1 w k) x = (w k x, w 'c' 'd')
```

Here we do not need to give a type signature to `w`, because it is an argument of constructor `T1` and that tells GHC all it needs to know.

6.4.19.4 Implicit quantification

GHC performs implicit quantification as follows. At the outermost level (only) of user-written types, if and only if there is no explicit `forall`, GHC finds all the type variables mentioned in the type that are not already in scope, and universally quantifies them. For example, the following pairs are equivalent:

```
f :: a -> a
f :: forall a. a -> a

g (x::a) = let
              h :: a -> b -> b
              h x y = y
           in ...
g (x::a) = let
              h :: forall b. a -> b -> b
              h x y = y
           in ...
```

Notice that GHC always adds implicit quantifiers *at the outermost level* of a user-written type; it does *not* find the inner-most possible quantification point. For example:

```
f :: (a -> a) -> Int
         -- MEANS
f :: forall a. (a -> a) -> Int
         -- NOT
f :: (forall a. a -> a) -> Int


g :: (Ord a => a -> a) -> Int
```

(continues on next page)

(continued from previous page)

```
        -- MEANS
g :: forall a. (Ord a => a -> a) -> Int
        -- NOT
g :: (forall a. Ord a => a -> a) -> Int
```

If you want the latter type, you can write your `forall`s explicitly. Indeed, doing so is strongly advised for rank-2 types.

Sometimes there *is* no "outermost level", in which case no implicit quantification happens:

```
data PackMap a b s t = PackMap (Monad f => (a -> f b) -> s -> f t)
```

This is rejected because there is no "outermost level" for the types on the RHS (it would obviously be terrible to add extra parameters to `PackMap`), so no implicit quantification happens, and the declaration is rejected (with "`f` is out of scope"). Solution: use an explicit `forall`:

```
data PackMap a b s t = PackMap (forall f. Monad f => (a -> f b) -> s -> f t)
```

6.4.20 Impredicative polymorphism

`ImpredicativeTypes`

Implies *`RankNTypes`* (page 402)

Since 9.2.1 (unreliable in 6.10 - 9.0)

Allow impredicative polymorphic types.

In general, GHC will only instantiate a polymorphic function at a monomorphic type (one with no foralls). For example,

```
runST :: (forall s. ST s a) -> a
id :: forall b. b -> b

foo = id runST   -- Rejected
```

The definition of `foo` is rejected because one would have to instantiate `id`'s type with `b := (forall s. ST s a) -> a`, and that is not allowed. Instantiating polymorphic type variables with polymorphic types is called *impredicative polymorphism*.

GHC has robust support for *impredicative polymorphism*, enabled with *`ImpredicativeTypes`* (page 407), using the so-called Quick Look inference algorithm. It is described in the paper A quick look at impredicativity (Serrano et al, ICFP 2020).

Switching on *`ImpredicativeTypes`* (page 407)

- Switches on *`RankNTypes`* (page 402)
- Allows user-written types to have foralls under type constructors, not just under arrows. For example `f :: Maybe (forall a. [a] -> [a])` is a legal type signature.
- Allows polymorphic types in Visible Type Application (when *`TypeApplications`* (page 386) is enabled). For example, you can write `reverse @(forall b. b->b) xs`. Using VTA with a polymorphic type argument is useful in cases when Quick Look cannot infer the correct instantiation.
- Switches on the Quick Look type inference algorithm, as described in the paper. This allows the compiler to infer impredicative instantiations of polymorphic functions in many

cases. For example, `reverse xs` will typecheck even if `xs :: [forall a. a->a]`, by instantiating `reverse` at type `forall a. a->a`.

Note that the treatment of type-class constraints and implicit parameters remains entirely monomorphic, even with ImpredicativeTypes. Specifically:

- You cannot apply a type class to a polymorphic type. This is illegal: `f :: C (forall a. a->a) => [a] -> [a]`
- You cannot give an instance declaration with a polymorphic argument. This is illegal: `instance C (forall a. a->a)`
- An implicit parameter cannot have a polymorphic type: `g :: (?x :: forall a. a->a) => [a] -> [a]`

For many years GHC has a special case for the function `($)`, that allows it to typecheck an application like `runST $ (do { ... })`, even though that instantiation may be impredicative. This special case remains: even without *ImpredicativeTypes* (page 407) GHC switches on Quick Look for applications of `($)`.

This flag was available in earlier versions of GHC (6.10.1 - 9.0), but the behavior was unpredictable and not officially supported.

6.4.21 Linear types

LinearTypes

Implies *MonoLocalBinds* (page 526)

Since 9.0.1

Status Experimental

Enable the linear arrow `a %1 -> b` and the multiplicity-polymorphic arrow `a %m -> b`.

This extension is currently considered experimental, expect bugs, warts, and bad error messages; everything down to the syntax is subject to change. See, in particular, *Limitations* (page 412) below. We encourage you to experiment with this extension and report issues in the GHC bug tracker, adding the tag `LinearTypes`.

A function `f` is linear if: when its result is consumed *exactly once*, then its argument is consumed *exactly once*. Intuitively, it means that in every branch of the definition of `f`, its argument `x` must be used exactly once. Which can be done by

- Returning `x` unmodified
- Passing `x` to a *linear* function
- Pattern-matching on `x` and using each argument exactly once in the same fashion.
- Calling it as a function and using the result exactly once in the same fashion.

With `-XLinearTypes`, you can write `f :: a %1 -> b` to mean that `f` is a linear function from `a` to `b`. If *UnicodeSyntax* (page 277) is enabled, the `%1 ->` arrow can be written as `⊸`.

To allow uniform handling of linear `a %1 -> b` and unrestricted `a -> b` functions, there is a new function type `a %m -> b`. Here, `m` is a type of new kind `Multiplicity`. We have:

```
data Multiplicity = One | Many  -- Defined in GHC.Types

type a %1 -> b = a %One  -> b
type a  -> b = a %Many -> b
```

(See *Datatype promotion* (page 357)).

We say that a variable whose multiplicity constraint is `Many` is *unrestricted*.

The multiplicity-polymorphic arrow `a %m -> b` is available in a prefix version as `GHC.Exts.FUN m a b`, which can be applied partially. See, however *Limitations* (page 412).

Linear and multiplicity-polymorphic arrows are *always declared*, never inferred. That is, if you don't give an appropriate type signature to a function, it will be inferred as being a regular function of type `a -> b`. The same principle holds for representation polymorphism (see *Inference and defaulting* (page 383)).

6.4.21.1 Expressions

When defining a function either as a lambda `\x -> u` or with equations `f x = u`, the multiplicity of the variable `x` will be inferred from the context. For equations, the context will typically be a type signature. For instance here is a linear function

```
f :: (a -> b) %1 -> a -> b
f g x = g x
```

In this example, `g` must be used linearly while `x` is unrestricted.

Bindings

Let and where bindings can be linear as well, the multiplicity of bindings is typically inferred

```
f :: A %1 -> B
g :: B %1 -> C

h :: A %1 -> C
h x = g y
  where
    y = f x
```

If you don't want, or aren't able, to rely on inference, let and where bindings can be annotated with a multiplicity

```
f :: A %1 -> B
g :: B %1 -> C

h :: A %1 -> C
h x = g y
  where
    %1 y = f x
```

The precise rules are, that you can annotate a binding with a multiplicity if:

- The binding is not top-level
- The binding is non-recursive
- The binding is a pattern binding (including a simple variable) `p=e` (you can't write `let %1 f x = u`, instead write `let %1 f = \x -> u`)
- Either `p` is strict (see infra) or `p` is a variable. In particular neither `x@y` nor `(x)` are covered by "is a variable"

When there's no multiplicity annotation, the multiplicity is inferred as follows:

- Toplevel bindings are inferred as having multiplicity `Many`
- Recursive bindings are inferred as having multiplicity `Many`
- Lazy non-variable pattern bindings are inferred as having multiplicity `Many` (note that in let- and where-bindings, patterns are lazy by default, so that `let (x,y) = rhs` always have multiplicity `Many`, whereas `let !(x,y) = rhs` can have multiplicity 1).
- In all other cases, including function bindings `let f x1...xn = rhs`, the multiplicity is inferred from the term.

When `-XMonoLocalBinds` is off, the following also holds:

- Multiplicity-annotated non-variable pattern-bindings (such as `let %1 !(x,y) = rhs`) are never generalised.
- Non-variable pattern bindings which are inferred as polymorphic or qualified are inferred as having multiplicity `Many`.

Strict patterns

GHC considers that non-variable lazy patterns consume the scrutinee with multiplicity `Many`. In practice, a pattern is strict (hence can be linear) if (otherwise the pattern is lazy):

- The pattern is a case alternative and isn't annotated with a ~
- The pattern is a let-binding, and is annotated with a `!`
- The pattern is a let-binding, *Strict* (page 543) is on, and isn't annotated with a ~
- The pattern is nested inside a strict pattern

Here are some examples of the impact on linear typing:

Without `-XStrict`:

```
-- good
let %1 x = u in …

-- good
let %1 !x = u in …

-- bad
let %1 (x, y) = u in …

-- good
let %Many (x, y) = u in …

-- good
let %1 !(x, y) = u in …

-- good
let %1 (!(x, y)) = u in …

-- inferred unrestricted
let (x, y) = u in …

-- can be inferred linear
case u of (x, y) -> …
```

(continues on next page)

(continued from previous page)

```
-- inferred unrestricted
case u of ~(x, y) -> …
```

With -XStrict:: – good let %1 x = u in …

– good let %1 !x = u in …

– good let %1 (x, y) = u in …

– bad let %1 ~(x, y) = u in …

– good let %Many ~(x, y) = u in …

– can be inferred linear let (x, y) = u in …

– inferred unrestricted let ~(x, y) = u in …

6.4.21.2 Data types

By default, all fields in algebraic data types are linear (even if `-XLinearTypes` is not turned on). Given

```
data T1 a = MkT1 a
```

the value `MkT1 x` can be constructed and deconstructed in a linear context:

```
construct :: a %1 -> T1 a
construct x = MkT1 x

deconstruct :: T1 a %1 -> a
deconstruct (MkT1 x) = x  -- must consume `x` exactly once
```

When used as a value, `MkT1` is given a multiplicity-polymorphic type: `MkT1 :: forall {m} a. a %m -> T1 a`. This makes it possible to use `MkT1` in higher order functions. The additional multiplicity argument `m` is marked as inferred (see *Inferred vs. specified type variables* (page 387)), so that there is no conflict with visible type application. When displaying types, unless `-XLinearTypes` is enabled, multiplicity polymorphic functions are printed as regular functions (see *Printing multiplicity-polymorphic types* (page 412)); therefore constructors appear to have regular function types.

```
mkList :: [a] -> [T1 a]
mkList xs = map MkT1 xs
```

Hence the linearity of type constructors is invisible when `-XLinearTypes` is off.

Whether a data constructor field is linear or not can be customized using the GADT syntax. Given

```
data T2 a b c where
    MkT2 :: a -> b %1 -> c %1 -> T2 a b c -- Note unrestricted arrow in the first␣
↪argument
```

the value `MkT2 x y z` can be constructed only if `x` is unrestricted. On the other hand, a linear function which is matching on `MkT2 x y z` must consume `y` and `z` exactly once, but there is no restriction on `x`.

It is also possible to define a multiplicity-polymorphic field:

```
data T3 a m where
    MkT3 :: a %m -> T3 a m
```

While linear fields are generalized (`MkT1 :: forall {m} a. a %m -> T1 a` in the previous example), multiplicity-polymorphic fields are not; it is not possible to directly use `MkT3` as a function `a -> T3 a One`.

If *LinearTypes* (page 408) is disabled, all fields are considered to be linear fields, including GADT fields defined with the `->` arrow.

In a `newtype` declaration, the field must be linear. Attempting to write an unrestricted newtype constructor with GADT syntax results in an error.

6.4.21.3 Printing multiplicity-polymorphic types

If *LinearTypes* (page 408) is disabled, multiplicity variables in types are defaulted to `Many` when printing, in the same manner as described in *Printing representation-polymorphic types* (page 384). In other words, without *LinearTypes* (page 408), multiplicity-polymorphic functions `a %m -> b` are printed as normal Haskell2010 functions `a -> b`. This allows existing libraries to be generalized to linear types in a backwards-compatible manner; the general types are visible only if the user has enabled *LinearTypes* (page 408). (Note that a library can declare a linear function in the contravariant position, i.e. take a linear function as an argument. In this case, linearity cannot be hidden; it is an essential part of the exposed interface.)

6.4.21.4 Limitations

Linear types are still considered experimental and come with several limitations. If you have read the full design in the proposal (see *Design and further reading* (page 413) below), here is a run down of the missing pieces.

- Multiplicity polymorphism is incomplete and experimental. You may have success using it, or you may not. Expect it to be really unreliable. (Multiplicity multiplication is not supported yet.)
- There is currently no support for multiplicity annotations on function arguments such as `\(%p x :: a) -> ...`, only on let-bound variables.
- A `case` expression may consume its scrutinee `One` time, or `Many` times. But the inference is still experimental, and may over-eagerly guess that it ought to consume the scrutinee `Many` times.
- There is no support for linear pattern synonyms.
- `@`-patterns and view patterns are not linear.
- The projection function for a record with a single linear field should be multiplicity-polymorphic; currently it's unrestricted.
- Attempting to use of linear types in Template Haskell will probably not work.

6.4.21.5 Design and further reading

- The design for this extension is described in details in the Linear types proposal
- This extension has been originally conceived of in the paper Linear Haskell: practical linearity in a higher-order polymorphic language (POPL 2018)
- There is a wiki page dedicated to the linear types extension

6.4.22 Custom compile-time errors

When designing embedded domain specific languages in Haskell, it is useful to have something like `error` at the type level. In this way, the EDSL designer may show a type error that is specific to the DSL, rather than the standard GHC type error.

For example, consider a type class that is not intended to be used with functions, but the user accidentally used it at a function type, perhaps because they missed an argument to some function. Then, instead of getting the standard GHC message about a missing instance, it would be nicer to emit a more friendly message specific to the EDSL. Similarly, the reduction of a type-level function may get stuck due to an error, at which point it would be nice to report an EDSL specific error, rather than a generic error about an ambiguous type.

To solve this, GHC provides a single type-level function,

```
type family TypeError (msg :: ErrorMessage) :: k
```

along with a small type-level language (via *DataKinds* (page 357)) for constructing pretty-printed error messages,

```
-- ErrorMessage is intended to be used as a kind
data ErrorMessage =
      Text Symbol                        -- Show this text as is
    | forall t. ShowType t               -- Pretty print a type
    | ErrorMessage :<>: ErrorMessage     -- Put two chunks of error message next to␣
↪each other
    | ErrorMessage :$$: ErrorMessage     -- Put two chunks of error message above␣
↪each other
```

in the GHC.TypeLits module.

For instance, we might use this interface to provide a more useful error message for applications of `show` on unsaturated functions like this,

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE UndecidableInstances #-}

import GHC.TypeLits

instance TypeError (Text "Cannot 'Show' functions." :$$:
                    Text "Perhaps there is a missing argument?")
         => Show (a -> b) where
   showsPrec = error "unreachable"

main = print negate
```

Which will produce the following compile-time error,

```
Test.hs:12:8: error:
    • Cannot 'Show' functions.
      Perhaps there is a missing argument?
    • In the expression: print negate
      In an equation for ‘main’: main = print negate
```

6.4.23 Deferring type errors to runtime

Since GHC 7.6.1.

While developing, sometimes it is desirable to allow compilation to succeed even if there are type errors in the code. Consider the following case:

```haskell
module Main where

a :: Int
a = 'a'

main = print "b"
```

Even though `a` is ill-typed, it is not used in the end, so if all that we're interested in is `main` it can be useful to be able to ignore the problems in `a`.

For more motivation and details please refer to the Wiki page or the original paper.

6.4.23.1 Enabling deferring of type errors

The flag *-fdefer-type-errors* (page 414) controls whether type errors are deferred to runtime. Type errors will still be emitted as warnings, but will not prevent compilation. You can use *-Wno-deferred-type-errors* (page 88) to suppress these warnings.

This flag implies the *-fdefer-typed-holes* (page 414) and *-fdefer-out-of-scope-variables* (page 415) flags, which enables this behaviour for *Typed Holes* (page 304) and variables. Should you so wish, it is possible to enable *-fdefer-type-errors* (page 414) without enabling *-fdefer-typed-holes* (page 414) or *-fdefer-out-of-scope-variables* (page 415), by explicitly specifying *-fno-defer-typed-holes* (page 414) or *-fno-defer-out-of-scope-variables* (page 415) on the command-line after the *-fdefer-type-errors* (page 414) flag.

-fdefer-type-errors

> **Since** 7.6
>
> **Implies** *-fdefer-typed-holes* (page 414), *-fdefer-out-of-scope-variables* (page 415)

Defer as many type errors as possible until runtime. At compile time you get a warning (instead of an error). At runtime, if you use a value that depends on a type error, you get a runtime error; but you can run any type-correct parts of your code just fine. See also *-Wdeferred-type-errors* (page 88).

-fdefer-typed-holes

> **Since** 7.10

Defer typed holes errors (errors about names with a leading underscore (e.g., "_", "_foo", "_bar")) until runtime. This will turn the errors produced by *typed holes* (page 304) into warnings. Using a value that depends on a typed hole produces a runtime error, the same as *-fdefer-type-errors* (page 414) (which implies this option). See *Typed Holes* (page 304).

Implied by *-fdefer-type-errors* (page 414). See also *-Wtyped-holes* (page 88).

-fdefer-out-of-scope-variables

Since 8.0

Defer variable out-of-scope errors (errors about names without a leading underscore) until runtime. This will turn variable-out-of-scope errors into warnings. Using a value that depends on an out-of-scope variable produces a runtime error, the same as *-fdefer-type-errors* (page 414) (which implies this option). See *Typed Holes* (page 304).

Implied by *-fdefer-type-errors* (page 414). See also *-Wdeferred-out-of-scope-variables* (page 88).

At runtime, whenever a term containing a type error would need to be evaluated, the error is converted into a runtime exception of type `TypeError`. Note that type errors are deferred as much as possible during runtime, but invalid coercions are never performed, even when they would ultimately result in a value of the correct type. For example, given the following code:

```
x :: Int
x = 0

y :: Char
y = x

z :: Int
z = y
```

evaluating `z` will result in a runtime `TypeError`.

6.4.23.2 Deferred type errors in GHCi

The flag *-fdefer-type-errors* (page 414) works in GHCi as well, with one exception: for "naked" expressions typed at the prompt, type errors don't get delayed, so for example:

```
Prelude> fst (True, 1 == 'a')

<interactive>:2:12:
    No instance for (Num Char) arising from the literal `1'
    Possible fix: add an instance declaration for (Num Char)
    In the first argument of `(==)', namely `1'
    In the expression: 1 == 'a'
    In the first argument of `fst', namely `(True, 1 == 'a')'
```

Otherwise, in the common case of a simple type error such as typing `reverse True` at the prompt, you would get a warning and then an immediately-following type error when the expression is evaluated.

This exception doesn't apply to statements, as the following example demonstrates:

```
Prelude> let x = (True, 1 == 'a')

<interactive>:3:16: Warning:
```

(continues on next page)

(continued from previous page)

```
    No instance for (Num Char) arising from the literal `1'
    Possible fix: add an instance declaration for (Num Char)
    In the first argument of `(==)', namely `1'
    In the expression: 1 == 'a'
    In the expression: (True, 1 == 'a')
Prelude> fst x
True
```

6.4.23.3 Limitations of deferred type errors

The errors that can be deferred are:

- Out of scope term variables
- Equality constraints; e.g. `ord True` gives rise to an insoluble equality constraint `Char ~ Bool`, which can be deferred.
- Type-class and implicit-parameter constraints

All other type errors are reported immediately, and cannot be deferred; for example, an ill-kinded type signature, an instance declaration that is non-terminating or ill-formed, a type-family instance that does not obey the declared injectivity constraints, etc etc.

In a few cases, some constraints cannot be deferred. Specifically:

- Kind errors in a type or kind signature, partial type signatures, or pattern signature. e.g.

  ```
  f :: Int Bool -> Char
  ```

  This type signature contains a kind error which cannot be deferred.
- Type equalities under a forall (c.f. #14605).
- Kind errors in a visible type application. e.g.

  ```
  reverse @Maybe xs
  ```

- Kind errors in a `default` declaration. e.g.

  ```
  default( Double, Int Int )
  ```

- Errors involving linear types (c.f. #20083). e.g.

  ```
  f :: a %1 -> a
  f _  = ()
  ```

- Illegal representation polymorphism, e.g.

  ```
  f :: forall rep (a :: TYPE rep). a -> a
  f a = a
  ```

6.4.24 Roles

Using *GeneralizedNewtypeDeriving* (page 447) (*Generalised derived instances for newtypes* (page 447)), a programmer can take existing instances of classes and "lift" these into instances of that class for a newtype. However, this is not always safe. For example, consider the following:

```
newtype Age = MkAge { unAge :: Int }

type family Inspect x
type instance Inspect Age = Int
type instance Inspect Int = Bool

class BadIdea a where
  bad :: a -> Inspect a

instance BadIdea Int where
  bad = (> 0)

deriving instance BadIdea Age    -- not allowed!
```

If the derived instance were allowed, what would the type of its method `bad` be? It would seem to be `Age -> Inspect Age`, which is equivalent to `Age -> Int`, according to the type family `Inspect`. Yet, if we simply adapt the implementation from the instance for `Int`, the implementation for `bad` produces a `Bool`, and we have trouble.

The way to identify such situations is to have *roles* assigned to type variables of datatypes, classes, and type synonyms.

Roles as implemented in GHC are a from a simplified version of the work described in Generative type abstraction and type-level computation, published at POPL 2011.

6.4.24.1 Nominal, Representational, and Phantom

The goal of the roles system is to track when two types have the same underlying representation. In the example above, `Age` and `Int` have the same representation. But, the corresponding instances of `BadIdea` would *not* have the same representation, because the types of the implementations of `bad` would be different.

Suppose we have two uses of a type constructor, each applied to the same parameters except for one difference. (For example, `T Age Bool c` and `T Int Bool c` for some type `T`.) The role of a type parameter says what we need to know about the two differing type arguments in order to know that the two outer types have the same representation (in the example, what must be true about `Age` and `Int` in order to show that `T Age Bool c` has the same representation as `T Int Bool c`).

GHC supports three different roles for type parameters: nominal, representational, and phantom. If a type parameter has a nominal role, then the two types that differ must not actually differ at all: they must be identical (after type family reduction). If a type parameter has a representational role, then the two types must have the same representation. (If `T`'s first parameter's role is representational, then `T Age Bool c` and `T Int Bool c` would have the same representation, because `Age` and `Int` have the same representation.) If a type parameter has a phantom role, then we need no further information.

Here are some examples:

```
data Simple a = MkSimple a          -- a has role representational

type family F
type instance F Int = Bool
type instance F Age = Char

data Complex a = MkComplex (F a)    -- a has role nominal

data Phant a = MkPhant Bool         -- a has role phantom
```

The type `Simple` has its parameter at role representational, which is generally the most common case. `Simple Age` would have the same representation as `Simple Int`. The type `Complex`, on the other hand, has its parameter at role nominal, because `Complex Age` and `Complex Int` are *not* the same. Lastly, `Phant Age` and `Phant Bool` have the same representation, even though `Age` and `Bool` are unrelated.

6.4.24.2 Role inference

What role should a given type parameter have? GHC performs role inference to determine the correct role for every parameter. It starts with a few base facts: `(->)` has two representational parameters; `(~)` has two nominal parameters; all type families' parameters are nominal; and all GADT-like parameters are nominal. Then, these facts are propagated to all places where these types are used. The default role for datatypes and synonyms is phantom; the default role for classes is nominal. Thus, for datatypes and synonyms, any parameters unused in the right-hand side (or used only in other types in phantom positions) will be phantom. Whenever a parameter is used in a representational position (that is, used as a type argument to a constructor whose corresponding variable is at role representational), we raise its role from phantom to representational. Similarly, when a parameter is used in a nominal position, its role is upgraded to nominal. We never downgrade a role from nominal to phantom or representational, or from representational to phantom. In this way, we infer the most-general role for each parameter.

Classes have their roles default to nominal to promote coherence of class instances. If a `C Int` were stored in a datatype, it would be quite bad if that were somehow changed into a `C Age` somewhere, especially if another `C Age` had been declared!

There is one particularly tricky case that should be explained:

```
data Tricky a b = MkTricky (a b)
```

What should `Tricky`'s roles be? At first blush, it would seem that both `a` and `b` should be at role representational, since both are used in the right-hand side and neither is involved in a type family. However, this would be wrong, as the following example shows:

```
data Nom a = MkNom (F a)   -- type family F from example above
```

Is `Tricky Nom Age` representationally equal to `Tricky Nom Int`? No! The former stores a `Char` and the latter stores a `Bool`. The solution to this is to require all parameters to type variables to have role nominal. Thus, GHC would infer role representational for `a` but role nominal for `b`.

6.4.24.3 Role annotations

RoleAnnotations

Since 7.8.1

Status Included in *GHC2024* (page 270)

Allow role annotation syntax.

Sometimes the programmer wants to constrain the inference process. For example, we can consider a type `Set a` that represents a set of data, ordered according to `a`'s `Ord` instance. While it would generally be type-safe to consider `a` to be at role representational, it is possible that a `newtype` and its base type have *different* orderings encoded in their respective `Ord` instances. This would lead to misbehavior at runtime. So, the author of the `Set` datatype would like its parameter to be at role nominal. This would be done with a declaration

```
type role Set nominal
```

Role annotations can also be used should a programmer wish to write a class with a representational (or phantom) role. However, as a class with non-nominal roles can quickly lead to class instance incoherence, it is necessary to also specify *IncoherentInstances* (page 486) to allow non-nominal roles for classes.

The other place where role annotations may be necessary are in `hs-boot` files (*How to compile mutually recursive modules* (page 207)), where the right-hand sides of definitions can be omitted. As usual, the types/classes declared in an `hs-boot` file must match up with the definitions in the `hs` file, including down to the roles. The default role for datatypes is representational in `hs-boot` files, corresponding to the common use case.

Role annotations are allowed on data, newtype, and class declarations. A role annotation declaration starts with `type role` and is followed by one role listing for each parameter of the type. (This parameter count includes parameters implicitly specified by a kind signature in a GADT-style data or newtype declaration.) Each role listing is a role (`nominal`, `representational`, or `phantom`) or a `_`. Using a `_` says that GHC should infer that role. The role annotation may go anywhere in the same module as the datatype or class definition (much like a value-level type signature). Here are some examples:

```
type role T1 _ phantom
data T1 a b = MkT1 a     -- b is not used; annotation is fine but unnecessary

type role T2 _ phantom
data T2 a b = MkT2 b     -- ERROR: b is used and cannot be phantom

type role T3 _ nominal
data T3 a b = MkT3 a     -- OK: nominal is higher than necessary, but safe

type role T4 nominal
data T4 a = MkT4 (a Int) -- OK, but nominal is higher than necessary

type role C representational _   -- OK, with -XIncoherentInstances
class C a b where ...    -- OK, b will get a nominal role

type role X nominal
type X a = ...           -- ERROR: role annotations not allowed for type synonyms
```

6.5 Records

6.5.1 Record field name resolution

A record field name *x* can be used in four contexts:

1. In a record *construction*: `C{ x = 3 }`
2. In a record *update*: `r{ x = 4 }`
3. In a record *pattern match*: `case r of C{ x = value } -> …`
4. As a standalone record *selector*: `x r`

In these four cases, here are the rules for field name resolution:

1. An unqualified name "x" is unambiguous *if and only if* there is just one "x" in scope unqualified.
2. A qualified name "M.x" is unambiguous *if and only if* there is just one "M.x" in scope.

Those rules are amended by the following extensions:

- *`DisambiguateRecordFields`* (page 423): In record construction and pattern matching (`C{ x = …}`), an unqualified field name `x` is unambiguous if and only if the data constructor (`C`) has a field `x`, and that field is in scope unqualified, or qualified as `Q.x`, regardless of `Q`. Similarly, in record construction and pattern matching, a qualified field name `M.x` is unambiguous if and only if the data constructor (`C`) has a field `x`, and that field is in scope qualified as `M.x`.

  In record updates (`r{ x = 3 }`), the field name x is unambiguous if and only if there is just one field name x in scope unqualified. Similarly, the record update with a qualified field `r{ M.x = 3 }` is unambiguous if just one field name is in scope as `M.x`. In both cases, non-field names are ignored.

- *`DuplicateRecordFields`* (page 425): This extension allows record updates if exactly one type has all the fields being updated, even if they are individually ambiguous according to the two rules for field name resolution above.

  For example:

```
data S = MkS1 { x :: Int, y :: Bool }
       | MkS2 { x :: Int }

data T = MkT1 { x :: Int, z :: Bool }
       | MkT2 { z :: Bool }

f r = r{ x=3, y=True }
```

The only data type that has both `x` and `y` as fields is `S`, so the field names `x` and `y` refer unambiguously to data type `S`.

- *`NoFieldSelectors`* (page 426): This extension being enabled means that field selector names in scope will be ignored in an expression context.

  For example:

```
data T = MkT { x :: Int }

x :: String
```

(continues on next page)

(continued from previous page)

```
x = "Hello"

f = x
```

With `NoFieldSelectors` the `x` in `f`'s right hand side refers to the `x :: String`, not to the field `x`.

6.5.2 Traditional record syntax

NoTraditionalRecordSyntax

Since 7.4.1

Disallow use of record syntax.

Traditional record syntax, such as `C {f = x}`, is enabled by default. To disable it, you can use the *NoTraditionalRecordSyntax* (page 421) extension.

Under *NoTraditionalRecordSyntax* (page 421), it is not permitted to define a record datatype or use record syntax in an expression. For example, the following all require *TraditionalRecordSyntax* (page 421):

```
data T = MkT { foo :: Int }  -- record datatype definition

x = MkT { foo = 3 }          -- construction

y = x { foo = 3 }            -- update

f (MkT { foo = i }) = i      -- pattern matching
```

However, if a field selector function is in scope, it may be used normally. (This arises if a module using *NoTraditionalRecordSyntax* (page 421) imports a module that defined a record with *TraditionalRecordSyntax* (page 421) enabled). If you wish to suppress field selector functions, use the *NoFieldSelectors* (page 426) extension.

6.5.3 Field selectors and TypeApplications

Field selectors can be used in conjunction with *TypeApplications* (page 386), as described in *Visible type application* (page 386). The type of a field selector is constructed by using the surrounding definition as context. This section provides a specification for how this construction works. We will explain it by considering three different forms of field selector, each of which is a minor variation of the same general theme.

6.5.3.1 Field selectors for Haskell98-style data constructors

Consider the following example:

```
data T a b = MkT { unT :: forall e. Either e a }
```

This data type uses a Haskell98-style declaration. The only part of this data type that is not Haskell98 code is `unT`, whose type uses higher-rank polymorphism (*Arbitrary-rank polymorphism* (page 402)). To construct the type of the `unT` field selector, we will assemble the following:

1. The type variables quantified by the data type head (`forall a b. <...>`).
2. The return type of the data constructor (`<...> T a b -> <...>`). By virtue of this being a Haskell98-style declaration, the order of type variables in the return type will always coincide with the order in which they are quantified.
3. The type of the field (`<...> forall e. Either e a`).

The final type of `unT` is therefore `forall a b. T a b -> forall e. Either e a`. As a result, one way to use `unT` with *TypeApplications* (page 386) is `unT @Int @Bool (MkT (Right 1)) @Char`.

6.5.3.2 Field selectors for GADT constructors

Field selectors for GADT constructors (*Declaring data types with explicit constructor signatures* (page 329)) are slightly more involved. Consider the following example:

```
data G a b where
  MkG :: forall x n a. (Eq a, Show n)
      => { unG1 :: forall e. Either e (a, x), unG2 :: n } -> G a (Maybe x)
```

The `MkG` GADT constructor has two records, `unG1` and `unG2`. However, only `unG1` can be used as a top-level field selector. `unG2` cannot because it is a "hidden" selector (see *Record Constructors* (page 326)); its type mentions a free variable `n` that does not appear in the result type `G a (Maybe x)`. On the other hand, the only free type variables in the type of `unG1` are `a` and `x`, so `unG1` is fine to use as a top-level function.

To construct the type of the `unG1` field selector, we will assemble the following:

1. The subset of type variables quantified by the GADT constructor that are mentioned in the return type. Note that the order of these variables follows the same principles as in *Ordering of specified variables* (page 388). If the constructor explicitly quantifies its type variables at the beginning of the type, then the field selector type will quantify them in the same order (modulo any variables that are dropped due to not being mentioned in the return type). If the constructor implicitly quantifies its type variables, then the field selector type will quantify them in the left-to-right order that they appear in the field itself.

   In this example, `MkG` explicitly quantifies `forall x n a.`, and of those type variables, `a` and `x` are mentioned in the return type. Therefore, the type of `unG1` starts as `forall x a. <...>`. If `MkG` had not used an explicit `forall`, then they would have instead been ordered as `forall a x. <...>`, since `a` appears to the left of `x` in the field type.
2. The GADT return type (`<...> G a (Maybe x) -> ...`).
3. The type of the field (`<...> -> forall e. Either e (a, x)`).

The final type of `unG1` is therefore `forall x a. G a (Maybe x) -> forall e. Either e (a, x)`. As a result, one way to use `unG1` with *TypeApplications* (page 386) is `unG1 @Int @Bool (MkG (Right (True, 42)) ()) @Char`.

6.5.3.3 Field selectors for pattern synonyms

Certain record pattern synonyms (*Record Pattern Synonyms* (page 464)) can give rise to top-level field selectors. Consider the following example:

```
pattern P :: forall a. Read a
          => forall n. (Eq a, Show n)
          => (forall e. Either e (a, Bool)) -> n -> G a (Maybe Bool)
pattern P {unP1, unP2} = MkG unP1 unP2
```

We can only make field selectors for pattern synonym records that do not mention any existential type variables whatsoever in their types, per *Record Pattern Synonyms* (page 464). (This is a stronger requirement than for GADT records, whose types can mention existential type variables provided that they are also mentioned in the return type.) We can see that `unP2` cannot be used as a top-level field selector since its type has a free type variable `n`, which is existential. `unP1` is fine, on the other hand, as its type only has one free variable, the universal type variable `a`.

To construct the type of the `unP1` field selector, we will assemble the following:

1. The universal type variables (`forall a. <...>`).
2. The required constraints (`<...> Read a => <...>`).
3. The pattern synonym return type (`<...> G a (Maybe Bool) -> <...>`).
4. The type of the field (`<...> -> forall e. Either e (a, Bool))`.

The final type of `unP1` is therefore `forall a. Read a => G a (Maybe Bool) -> forall e. Either e (a, Bool)`. As a result, one way to use `unP1` with *`TypeApplications`* (page 386) is `unP1 @Double (MkG (Right (4.5, True)) ()) @Char`.

6.5.4 Record field disambiguation

`DisambiguateRecordFields`

Since 6.8.1

Implied by *`RecordWildCards`* (page 428), *`DuplicateRecordFields`* (page 425)

Status Included in *`GHC2024`* (page 270)

Allow the compiler to automatically choose between identically-named record fields (if the choice is unambiguous).

In record construction and record pattern matching it is entirely unambiguous which field is referred to, even if there are two different data types in scope with a common field name. For example:

```
module M where
  data S = MkS { x :: Int, y :: Bool }

module Foo where
  import M

  data T = MkT { x :: Int }

  ok1 (MkS { x = n }) = n+1   -- Unambiguous
  ok2 n = MkT { x = n+1 }     -- Unambiguous
```

(continues on next page)

(continued from previous page)

```
bad1 k = k { x = 3 }        -- Ambiguous
bad2 k = x k                -- Ambiguous
```

Even though there are two x's in scope, it is clear that the x in the pattern in the definition of ok1 can only mean the field x from type S. Similarly for the function ok2. However, in the record update in bad1 and the record selection in bad2 it is not clear which of the two types is intended.

Haskell 98 regards all four as ambiguous, but with the *DisambiguateRecordFields* (page 423) extension, GHC will accept the former two. The rules are precisely the same as those for instance declarations in Haskell 98, where the method names on the left-hand side of the method bindings in an instance declaration refer unambiguously to the method of that class (provided they are in scope at all), even if there are other variables in scope with the same name. This reduces the clutter of qualified names when you import two records from different modules that use the same field name.

Since version 9.2.1, record fields in updates are disambiguated by ignoring non-field names in scope. For example, the following is accepted under *DisambiguateRecordFields* (page 423):

```
module Bar where
  import M  -- imports the field x

  x = ()

  e r = r { x = 0 }  -- unambiguously refers to the field
```

Some details:

- Field disambiguation can be combined with punning (see *Record puns* (page 427)). For example:

  ```
  module Foo where
    import M
    x=True
    ok3 (MkS { x }) = x+1   -- Uses both disambiguation and punning
  ```

- With *DisambiguateRecordFields* (page 423) you can use *unqualified* field names even if the corresponding selector is only in scope *qualified* For example, assuming the same module M as in our earlier example, this is legal:

  ```
  module Foo where
    import qualified M    -- Note qualified

    ok4 (M.MkS { x = n }) = n+1   -- Unambiguous
  ```

  Since the constructor MkS is only in scope qualified, you must name it M.MkS, but the field x does not need to be qualified even though M.x is in scope but x is not (In effect, it is qualified by the constructor).

6.5.5 Duplicate record fields

DuplicateRecordFields

Implies *DisambiguateRecordFields* (page 423)

Since 8.0.1

Allow definition of record types with identically-named fields.

Going beyond *DisambiguateRecordFields* (page 423) (see *Record field disambiguation* (page 423)), the *DuplicateRecordFields* (page 425) extension allows multiple datatypes to be declared using the same field names in a single module. For example, it allows this:

```
module M where
  data S = MkS { x :: Int }
  data T = MkT { x :: Bool }
```

Uses of fields that are always unambiguous because they mention the constructor, including construction and pattern-matching, may freely use duplicated field names. For example, the following are permitted (just as with *DisambiguateRecordFields* (page 423)):

```
s = MkS { x = 3 }

f (MkT { x = b }) = b
```

While *DuplicateRecordFields* (page 425) permits multiple fields with the same name in a single module, it does not permit a field and a normal value binding to have the same name. For that, use *NoFieldSelectors* (page 426).

As of GHC 9.4.1, selector names have to be entirely unambiguous (under the usual name resolution rules), while for record updates, there must be at most one datatype that has all the field names being updated.

6.5.5.1 Import and export of record fields

When *DuplicateRecordFields* (page 425) is enabled, an ambiguous field must be exported as part of its datatype, rather than at the top level. For example, the following is legal:

```
module M
  ( S(x)
  , T(..)
  ) where

data S = MkS { x :: Int }

data T = MkT { x :: Bool }
```

However, this would not be permitted, because `x` is ambiguous:

```
module M (x) where ...
```

For `import` statements, it is possible to import multiple fields with the same name, as well as importing individual fields as part of their datatypes. For example, the following imports are allowed:

```
import M (S(x))         -- imports the type S and the 'x' field of S (but not the␣
↪field of T)
import M (x)            -- imports both 'x' fields
import M hiding (S(x)) -- imports everything except the type S and its 'x' field
import M hiding (x)     -- imports everything except the two 'x' fields
```

6.5.6 Field selectors

FieldSelectors

> **Since** 9.2.1
>
> **Status** Included in *GHC2024* (page 270), *GHC2021* (page 271), *Haskell2010* (page 272), *Haskell98* (page 272)

Make record field selector functions visible in expressions.

By default, the *FieldSelectors* (page 426) extension is enabled, so defining a record datatype brings a selector function into scope for each field in the record. *NoFieldSelectors* (page 426) negates this feature, making it possible to:

- declare a top-level binding with the same name as a field, and
- refer to this top-level binding unambiguously in expressions.

Field labels are still usable within record construction, updates and pattern matching.

For example, given a datatype definition

```
data Foo = MkFoo { bar :: Int, baz :: String }
```

The following will be available:

1. the type constructor `Foo`;
2. the data constructor `MkFoo`;
3. the fields `bar` and `baz` for record construction, update, and pattern matching; and
4. the selector functions `bar :: Foo -> Int` and `baz :: Foo -> String`.

If the *NoFieldSelectors* (page 426) extension is enabled at the datatype definition site, items (1), (2), and (3) will still be available, but (4) will not. Correspondingly, it is permitted to define a top-level binding with the same name as a field, and using this name in an expression unambiguously refers to the non-field. For exmaple, the following is permitted:

```
data Foo = MkFoo { bar :: Int, baz :: String }
bar = ()  -- does not conflict with `bar` field
baz = bar -- unambiguously refers to `bar` the unit value, not the field
```

If you have multiple datatypes with the same field name, you need to enable *DuplicateRecordFields* (page 425) to allow them to be declared simultaneously. It is never permitted for a single module to define multiple top-level bindings with the same name.

The *DisambiguateRecordFields* (page 423) extension (implied by *DuplicateRecordFields* (page 425)) is useful in conjunction with *NoFieldSelectors* (page 426), because it excludes non-fields from consideration when resolving field names in record construction, update and pattern matching.

6.5.6.1 Import and export of selector functions

Under *FieldSelectors* (page 426), these modules are equivalent:

```
module A (Foo(MkFoo, bar, baz)) where
  data Foo = MkFoo { bar :: Int, baz :: Int }

module B (Foo(MkFoo, bar), baz) where
  data Foo = MkFoo { bar :: Int, baz :: Int }
```

Under *NoFieldSelectors* (page 426), these two export statements are now different. The first one will export the field `baz`, but not the top-level binding `baz`, while the second one would export the top-level binding `baz` (if one were defined), but not the field `baz`.

Because of this change, using *NoFieldSelectors* (page 426) and writing out selector functions explicitly is different to using *FieldSelectors* (page 426): in the former case the fields and functions must be exported separately. For example, here the selector functions are not exported:

```
{-# LANGUAGE NoFieldSelectors #-}
module M (Foo(MkFoo, bar, baz)) where
  data Foo = MkFoo { bar :: Int, baz :: Int }

  bar (MkFoo x _) = x
  baz (MkFoo _ x) = x
```

whereas here the selector functions are exported:

```
{-# LANGUAGE FieldSelectors #-}
module M (Foo(MkFoo, bar, baz)) where
  data Foo = MkFoo { bar :: Int, baz :: Int }
```

Wildcard exports will export the field labels, but will not export a top-level binding that happens to have the same name. In the examples above, exporting `Foo(..)` is (still) equivalent to exporting `Foo(MkFoo, bar, baz)`.

6.5.7 Record puns

NamedFieldPuns

Since 6.10.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow use of record puns.

Record puns are enabled by the language extension *NamedFieldPuns* (page 427).

When using records, it is common to write a pattern that binds a variable with the same name as a record field, such as:

```
data C = C {a :: Int}
f (C {a = a}) = a
```

Record punning permits the variable name to be elided, so one can simply write

```
f (C {a}) = a
```

to mean the same pattern as above. That is, in a record pattern, the pattern `a` expands into the pattern `a = a` for the same name `a`.

Note that:

- Record punning can also be used in an expression, writing, for example,

  ```
  let a = 1 in C {a}
  ```

  instead of

  ```
  let a = 1 in C {a = a}
  ```

  The expansion is purely syntactic, so the expanded right-hand side expression refers to the nearest enclosing variable that is spelled the same as the field name.

- Puns and other patterns can be mixed in the same record:

  ```
  data C = C {a :: Int, b :: Int}
  f (C {a, b = 4}) = a
  ```

- Puns can be used wherever record patterns occur (e.g. in `let` bindings or at the top-level).

- A pun on a qualified field name is expanded by stripping off the module qualifier. For example:

  ```
  f (C {M.a}) = a
  ```

  means

  ```
  f (M.C {M.a = a}) = a
  ```

  (This is useful if the field selector `a` for constructor `M.C` is only in scope in qualified form.)

6.5.8 Record wildcards

RecordWildCards

Implies *DisambiguateRecordFields* (page 423).

Since 6.8.1

Allow the use of wildcards in record construction and pattern matching.

Record wildcards are enabled by the language extension *RecordWildCards* (page 428). This extension implies *DisambiguateRecordFields* (page 423).

For records with many fields, it can be tiresome to write out each field individually in a record pattern, as in

```
data C = C {a :: Int, b :: Int, c :: Int, d :: Int}
f (C {a = 1, b = b, c = c, d = d}) = b + c + d
```

Record wildcard syntax permits a "`..`" in a record pattern, where each elided field `f` is replaced by the pattern `f = f`. For example, the above pattern can be written as

```
f (C {a = 1, ..}) = b + c + d
```

More details:

- Record wildcards in patterns can be mixed with other patterns, including puns (*Record puns* (page 427)); for example, in a pattern `(C {a = 1, b, ..})`. Additionally, record wildcards can be used wherever record patterns occur, including in `let` bindings and at the top-level. For example, the top-level binding

  ```
  C {a = 1, ..} = e
  ```

  defines `b`, `c`, and `d`.

- Record wildcards can also be used in an expression, when constructing a record. For example,

  ```
  let {a = 1; b = 2; c = 3; d = 4} in C {..}
  ```

  in place of

  ```
  let {a = 1; b = 2; c = 3; d = 4} in C {a=a, b=b, c=c, d=d}
  ```

  The expansion is purely syntactic, so the record wildcard expression refers to the nearest enclosing variables that are spelled the same as the omitted field names.

- For both pattern and expression wildcards, the "`..`" expands to the missing *in-scope* record fields. Specifically the expansion of "`C {..}`" includes `f` if and only if:
  - `f` is a record field of constructor `C`.
  - The record field `f` is in scope somehow (either qualified or unqualified).

  These rules restrict record wildcards to the situations in which the user could have written the expanded version. For example

  ```
  module M where
    data R = R { a,b,c :: Int }
  module X where
    import M( R(R,a,c) )
    f a b = R { .. }
  ```

  The `R{..}` expands to `R{a=a}`, omitting `b` since the record field is not in scope, and omitting `c` since the variable `c` is not in scope (apart from the binding of the record selector `c`, of course).

- When record wildcards are used in record construction, a field `f` is initialised only if `f` is in scope, and is not imported or bound at top level. For example, `f` can be bound by an enclosing pattern match or let/where-binding. For example

  ```
  module M where
    import A( a )

    data R = R { a,b,c,d :: Int }

    c = 3 :: Int

    f b = R { .. }  -- Expands to R { b = b, d = d }
      where
        d = b+1
  ```

  Here, `a` is imported, and `c` is bound at top level, so neither contribute to the expansion of the "`..`". The motivation here is that it should be easy for the reader to figure out what the "`..`" expands to.

- Record wildcards cannot be used (a) in a record update construct, and (b) for data constructors that are not declared with record fields. For example:

  ```
  f x = x { v=True, .. }   -- Illegal (a)

  data T = MkT Int Bool
  g = MkT { .. }           -- Illegal (b)
  h (MkT { .. }) = True    -- Illegal (b)
  ```

6.5.9 Record field selector polymorphism

The module GHC.Records defines the following:

```
class HasField (x :: k) r a | x r -> a where
  getField :: r -> a
```

A `HasField x r a` constraint represents the fact that `x` is a field of type `a` belonging to a record type `r`. The `getField` method gives the record selector function.

This allows definitions that are polymorphic over record types with a specified field. For example, the following works with any record type that has a field `name :: String`:

```
foo :: HasField "name" r String => r -> String
foo r = reverse (getField @"name" r)
```

`HasField` is a magic built-in typeclass (similar to `Coercible`, for example). It is given special treatment by the constraint solver (see *Solving HasField constraints* (page 430)). Users may define their own instances of `HasField` also (see *Virtual record fields* (page 432)).

6.5.9.1 Solving HasField constraints

If the constraint solver encounters a constraint `HasField x r a` where `r` is a concrete datatype with a field `x` in scope, it will automatically solve the constraint using the field selector as the dictionary, unifying `a` with the type of the field if necessary. This happens irrespective of which extensions are enabled.

For example, if the following datatype is in scope

```
data Person = Person { name :: String }
```

the end result is rather like having an instance

```
instance HasField "name" Person String where
  getField = name
```

except that this instance is not actually generated anywhere, rather the constraint is solved directly by the constraint solver.

A field must be in scope for the corresponding `HasField` constraint to be solved. This retains the existing representation hiding mechanism, whereby a module may choose not to export a field, preventing client modules from accessing or updating it directly.

Solving `HasField` constraints depends on the field selector functions that are generated for each datatype definition:

- If a record field does not have a selector function because its type would allow an existential variable to escape, the corresponding `HasField` constraint will not be solved. For example,

  ```
  {-# LANGUAGE ExistentialQuantification #-}
  data Exists t = forall x . MkExists { unExists :: t x }
  ```

  does not give rise to a selector `unExists :: Exists t -> t x` and we will not solve `HasField "unExists" (Exists t) a` automatically.

- If a record field has a polymorphic type (and hence the selector function is higher-rank), the corresponding `HasField` constraint will not be solved, because doing so would violate the functional dependency on `HasField` and/or require impredicativity. For example,

  ```
  {-# LANGUAGE RankNTypes #-}
  data Higher = MkHigher { unHigher :: forall t . t -> t }
  ```

  gives rise to a selector `unHigher :: Higher -> (forall t . t -> t)` but does not lead to solution of the constraint `HasField "unHigher" Higher a`.

- A record GADT may have a restricted type for a selector function, which may lead to additional unification when solving `HasField` constraints. For example,

  ```
  {-# LANGUAGE GADTs #-}
  data Gadt t where
    MkGadt :: { unGadt :: Maybe v } -> Gadt [v]
  ```

  gives rise to a selector `unGadt :: Gadt [v] -> Maybe v`, so the solver will reduce the constraint `HasField "unGadt" (Gadt t) b` by unifying `t ~ [v]` and `b ~ Maybe v` for some fresh metavariable `v`, rather as if we had an instance

  ```
  instance (t ~ [v], b ~ Maybe v) => HasField "unGadt" (Gadt t) b
  ```

- If a record type has an old-fashioned datatype context, the `HasField` constraint will be reduced to solving the constraints from the context. For example,

  ```
  {-# LANGUAGE DatatypeContexts #-}
  data Eq a => Silly a = MkSilly { unSilly :: a }
  ```

  gives rise to a selector `unSilly :: Eq a => Silly a -> a`, so the solver will reduce the constraint `HasField "unSilly" (Silly a) b` to `Eq a` (and unify `a` with `b`), rather as if we had an instance

  ```
  instance (Eq a, a ~ b) => HasField "unSilly" (Silly a) b
  ```

See *Overloaded record dot* (page 433) for an application of solving `HasField` constraints to implementing "record dot syntax".

6.5.9.2 Virtual record fields

Users may define their own instances of HasField, provided they do not conflict with the built-in constraint solving behaviour. This allows "virtual" record fields to be defined for datatypes that do not otherwise have them.

For example, this instance would make the name field of Person accessible using #fullname as well:

```
instance HasField "fullname" Person String where
  getField = name
```

More substantially, an anonymous records library could provide HasField instances for its anonymous records, and thus be compatible with the polymorphic record selectors introduced by this proposal. For example, something like this makes it possible to use getField to access Record values with the appropriate string in the type-level list of fields:

```
data Record (xs :: [(k, Type)]) where
  Nil  :: Record '[]
  Cons :: Proxy x -> a -> Record xs -> Record ('(x, a) ': xs)

instance HasField x (Record ('(x, a) ': xs)) a where
  getField (Cons _ v _) = v
instance HasField x (Record xs) a => HasField x (Record ('(y, b) ': xs)) a where
  getField (Cons _ _ r) = getField @x r

r :: Record '[ '("name", String) ]
r = Cons Proxy "R" Nil

x = getField @"name" r
```

Since representations such as this can support field labels with kinds other than Symbol, the HasField class is poly-kinded (even though the built-in constraint solving works only at kind Symbol). In particular, this allows users to declare scoped field labels such as in the following example:

```
data PersonFields = Name

s :: Record '[ '(Name, String) ]
s = Cons Proxy "S" Nil

y = getField @Name s
```

In order to avoid conflicting with the built-in constraint solving, the following user-defined HasField instances are prohibited (in addition to the usual rules, such as the prohibition on type families appearing in instance heads):

- `HasField _ r _` where r is a variable;
- `HasField _ (T ...) _` if T is a data family (because it might have fields introduced later, using data instance declarations);
- `HasField x (T ...) _` if x is a variable and T has any fields at all (but this instance is permitted if T has no fields);
- `HasField "foo" (T ...) _` if T has a field foo (but this instance is permitted if it does not).

If a field has a higher-rank or existential type, the corresponding HasField constraint will not be solved automatically (as described above), but in the interests of simplicity we do not permit users to define their own instances either. If a field is not in scope, the corresponding instance is still prohibited, to avoid conflicts in downstream modules.

6.5.10 Overloaded record dot

OverloadedRecordDot

Since 9.2.0

Provides record '.' syntax e.g. `x.foo`

When `OverloadedRecordDot` is enabled one can write `a.b` to mean the `b` field of the `a` record expression.

Example:

```
{-# LANGUAGE OverloadedRecordDot #-}
{-# LANGUAGE DuplicateRecordFields #-}

data Person = Person { name :: String }
data Company = Company { name :: String, owner :: Person }

main = do
  let c = Company { name = "Acme Corp."
                  , owner = Person { name = "Wile E. Coyote" } }
  print $ c.name ++ " is run by " ++ c.owner.name
```

You may also write `(.b)` to mean a function that "projects the `b` field from its argument". For example, `(.b) a` means the same thing as `a.b`).

`OverloadedRecordDot` is normally implemented by desugaring record `.` expressions to `GHC.Records.getField` expressions. By enabling `OverloadedRecordDot` and `RebindableSyntax` together it is possible to desugar `.` expressions into your own `getField` implementations.

When considering `a.b`, the `b` field that is meant is determined by solving `HasField` constraints. See *Solving HasField constraints* (page 430).

6.5.11 Overloaded record update

OverloadedRecordUpdate

Since 9.2.0

Provides record '.' syntax in record updates e.g. `x{foo.bar = 1}`.

EXPERIMENTAL *This design of this extension may well change in the future. It would be inadvisable to start using this extension for long-lived libraries just yet.*

It's usual (but not required) that this extension be used in conjunction with *Overloaded record dot* (page 433).

Example:

```
{-# LANGUAGE AllowAmbiguousTypes, FunctionalDependencies, ScopedTypeVariables,␣
↪PolyKinds, TypeApplications, DataKinds, FlexibleInstances #-}
{-# LANGUAGE NamedFieldPuns, RecordWildCards #-}
```

(continues on next page)

(continued from previous page)

```
{-# LANGUAGE OverloadedRecordDot, OverloadedRecordUpdate, RebindableSyntax #-}

import Prelude

class HasField x r a | x r -> a where
  hasField :: r -> (a -> r, a)

getField :: forall x r a . HasField x r a => r -> a
getField = snd . hasField @x -- Note: a.x = is getField @"x" a.
setField :: forall x r a . HasField x r a => r -> a -> r
setField = fst . hasField @x -- Note : a{x = b} is setField @"x" a b.

data Person = Person { name :: String } deriving Show
instance HasField "name" Person String where
    hasField r = (\x -> case r of Person { .. } -> Person { name = x, .. }, name r)

data Company = Company { company :: String, owner :: Person } deriving Show
instance HasField "company" Company String where
    hasField r = (\x -> case r of Company { .. } -> Company { company = x, .. }, company r)
instance HasField "owner" Company Person where
    hasField r = (\x -> case r of Company { .. } -> Company { owner = x, .. }, owner r)

main = do
  let c = Company {company = "Acme Corp.", owner = Person { name = "Wile E. Coyote" }}

  -- Top-level update
  print $ c{company = "Acme United"} -- Company {company = "Acme United", owner = Person {name = "Wile E. Coyote"}}

  -- Nested update
  print $ c{owner.name = "Walter C. Johnsen"} -- Company {company = "Acme Corp.", owner = Person {name = "Walter C. Johnsen"}}

  -- Punned update
  let name = "Walter C. Johnsen"
  print $ c{owner.name}  -- Company {company = "Acme Corp.", owner = Person {name = "Walter C. Johnsen"}}
```

`OverloadedRecordUpdate` works by desugaring record `.` update expressions to expressions involving the functions `setField` and `getField`. Note that all record updates will be desugared to `setField` expressions whether they use `.` notation or not.

At this time, `RebindableSyntax` must be enabled when `OverloadedRecordUpdate` is and users are required to provide definitions for `getField` and `setField`. We anticipate this restriction to be lifted in a future release of GHC with builtin support for `setField`.

6.6 Deriving mechanism

Haskell 98 allows the programmer to add a deriving clause to a data type declaration, to generate a standard instance declaration for specified class. GHC extends this mechanism along several axes:

- The derivation mechanism can be used separately from the data type declaration, using the *standalone deriving mechanism* (page 436).
- In Haskell 98, the only derivable classes are `Eq`, `Ord`, `Enum`, `Ix`, `Bounded`, `Read`, and `Show`. *Various language extensions* (page 438) extend this list.
- Besides the stock approach to deriving instances by generating all method definitions, GHC supports two additional deriving strategies, which can derive arbitrary classes:
  - *Generalised newtype deriving* (page 447) for newtypes and
  - *deriving any class* (page 453) using an empty instance declaration.

  The user can optionally declare the desired *deriving strategy* (page 455), especially if the compiler chooses the wrong one *by default* (page 456).

6.6.1 Deriving instances for empty data types

EmptyDataDeriving

Since 8.4.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271), *Haskell2010* (page 272)

Allow deriving instances of standard type classes for empty data types.

One can write data types with no constructors using the *EmptyDataDecls* (page 321) flag (see *Data types with no constructors* (page 321)), which is on by default in Haskell 2010. What is not on by default is the ability to derive type class instances for these types. This ability is enabled through use of the *EmptyDataDeriving* (page 435) flag. For instance, this lets one write:

```
data Empty deriving (Eq, Ord, Read, Show)
```

This would generate the following instances:

```
instance Eq Empty where
  _ == _ = True

instance Ord Empty where
  compare _ _ = EQ

instance Read Empty where
  readPrec = pfail

instance Show Empty where
  showsPrec _ x = case x of {}
```

The *EmptyDataDeriving* (page 435) flag is only required to enable deriving of these four "standard" type classes (which are mentioned in the Haskell Report). Other extensions to the `deriving` mechanism, which are explained below in greater detail, do not require

EmptyDataDeriving (page 435) to be used in conjunction with empty data types. These include:

- *StandaloneDeriving* (page 436) (see *Stand-alone deriving declarations* (page 436))
- Type classes which require their own extensions to be enabled to be derived, such as *DeriveFunctor* (page 439) (see *Deriving instances of extra classes (Data, etc.)* (page 438))
- *DeriveAnyClass* (page 453) (see *Deriving any other class* (page 453))

6.6.2 Inferred context for deriving clauses

The Haskell Report is vague about exactly when a `deriving` clause is legal. For example:

```
data T0 f a = MkT0 a         deriving( Eq )
data T1 f a = MkT1 (f a)     deriving( Eq )
data T2 f a = MkT2 (f (f a)) deriving( Eq )
```

The natural generated `Eq` code would result in these instance declarations:

```
instance Eq a         => Eq (T0 f a) where ...
instance Eq (f a)     => Eq (T1 f a) where ...
instance Eq (f (f a)) => Eq (T2 f a) where ...
```

The first of these is obviously fine. The second is still fine, although less obviously. The third is not Haskell 98, and risks losing termination of instances.

GHC takes a conservative position: it accepts the first two, but not the third. The rule is this: each constraint in the inferred instance context must consist only of type variables, with no repetitions.

This rule is applied regardless of flags. If you want a more exotic context, you can write it yourself, using the *standalone deriving mechanism* (page **??**).

6.6.3 Stand-alone deriving declarations

`StandaloneDeriving`

> **Since** 6.8.1
>
> **Status** Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow the use of stand-alone `deriving` declarations.

GHC allows stand-alone `deriving` declarations, enabled by *StandaloneDeriving* (page 436):

```
data Foo a = Bar a | Baz String

deriving instance Eq a => Eq (Foo a)
```

The syntax is identical to that of an ordinary instance declaration apart from (a) the keyword `deriving`, and (b) the absence of the `where` part.

However, standalone deriving differs from a `deriving` clause in a number of important ways:

- The standalone deriving declaration does not need to be in the same module as the data type declaration. (But be aware of the dangers of orphan instances (*Orphan modules and instance declarations* (page 218)).

- In most cases, you must supply an explicit context (in the example the context is `(Eq a)`), exactly as you would in an ordinary instance declaration. (In contrast, in a `deriving` clause attached to a data type declaration, the context is inferred.)

  The exception to this rule is that the context of a standalone deriving declaration can infer its context when a single, extra-wildcards constraint is used as the context, such as in:

```
deriving instance _ => Eq (Foo a)
```

  This is essentially the same as if you had written `deriving Eq` after the declaration for `data Foo a`. Using this feature requires the use of *`PartialTypeSignatures`* (page 520) (*Partial Type Signatures* (page 520)).

- Unlike a `deriving` declaration attached to a `data` declaration, the instance can be more specific than the data type (assuming you also use *`FlexibleInstances`* (page 480), *Instance termination rules* (page 483)). Consider for example

```
data Foo a = Bar a | Baz String

deriving instance Eq a => Eq (Foo [a])
deriving instance Eq a => Eq (Foo (Maybe a))
```

  This will generate a derived instance for `(Foo [a])` and `(Foo (Maybe a))`, but other types such as `(Foo (Int,Bool))` will not be an instance of `Eq`.

- Unlike a `deriving` declaration attached to a `data` declaration, GHC does not restrict the form of the data type. Instead, GHC simply generates the appropriate boilerplate code for the specified class, and typechecks it. If there is a type error, it is your problem. (GHC will show you the offending code if it has a type error.)

  The merit of this is that you can derive instances for GADTs and other exotic data types, providing only that the boilerplate code does indeed typecheck. For example:

```
data T a where
   T1 :: T Int
   T2 :: T Bool

deriving instance Show (T a)
```

  In this example, you cannot say `... deriving( Show )` on the data type declaration for `T`, because `T` is a GADT, but you *can* generate the instance declaration using stand-alone deriving.

  The down-side is that, if the boilerplate code fails to typecheck, you will get an error message about that code, which you did not write. Whereas, with a `deriving` clause the side-conditions are necessarily more conservative, but any error message may be more comprehensible.

- Under most circumstances, you cannot use standalone deriving to create an instance for a data type whose constructors are not all in scope. This is because the derived instance would generate code that uses the constructors behind the scenes, which would break abstraction.

  The one exception to this rule is *`DeriveAnyClass`* (page 453), since deriving an instance via *`DeriveAnyClass`* (page 453) simply generates an empty instance declaration, which does not require the use of any constructors. See the *deriving any class* (page **??**) section for more details.

In other ways, however, a standalone deriving obeys the same rules as ordinary deriving:

- A `deriving instance` declaration must obey the same rules concerning form and termination as ordinary instance declarations, controlled by the same flags; see *Instance declarations and resolution* (page 480).
- The stand-alone syntax is generalised for newtypes in exactly the same way that ordinary `deriving` clauses are generalised (*Generalised derived instances for newtypes* (page 447)). For example:

```
newtype Foo a = MkFoo (State Int a)

deriving instance MonadState Int Foo
```

  GHC always treats the *last* parameter of the instance (`Foo` in this example) as the type whose instance is being derived.

6.6.4 Deriving instances of extra classes (Data, etc.)

Haskell 98 allows the programmer to add "`deriving( Eq, Ord )`" to a data type declaration, to generate a standard instance declaration for classes specified in the `deriving` clause. In Haskell 98, the only classes that may appear in the `deriving` clause are the standard classes `Eq`, `Ord`, `Enum`, `Ix`, `Bounded`, `Read`, and `Show`.

GHC extends this list with several more classes that may be automatically derived:

- With *`DeriveGeneric`* (page 598), you can derive instances of the classes `Generic` and `Generic1`, defined in `GHC.Generics`. You can use these to define generic functions, as described in *Generic programming* (page 597).
- With *`DeriveFunctor`* (page 439), you can derive instances of the class `Functor`, defined in `GHC.Base`.
- With *`DeriveDataTypeable`* (page 445), you can derive instances of the class `Data`, defined in `Data.Data`.
- With *`DeriveFoldable`* (page 442), you can derive instances of the class `Foldable`, defined in `Data.Foldable`.
- With *`DeriveTraversable`* (page 444), you can derive instances of the class `Traversable`, defined in `Data.Traversable`. Since the `Traversable` instance dictates the instances of `Functor` and `Foldable`, you'll probably want to derive them too, so *`DeriveTraversable`* (page 444) implies *`DeriveFunctor`* (page 439) and *`DeriveFoldable`* (page 442).
- With *`DeriveLift`* (page 446), you can derive instances of the class `Lift`, defined in the `Language.Haskell.TH.Syntax` module of the `template-haskell` package.

You can also use a standalone deriving declaration instead (see *Stand-alone deriving declarations* (page 436)).

In each case the appropriate class must be in scope before it can be mentioned in the `deriving` clause.

6.6.4.1 Deriving `Functor` instances

DeriveFunctor

Implied by *DeriveTraversable* (page 444)

Since 7.10.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow automatic deriving of instances for the `Functor` typeclass.

With *DeriveFunctor* (page 439), one can derive `Functor` instances for data types of kind `Type -> Type`. For example, this declaration:

```
data Example a = Ex a Char (Example a) (Example Char)
  deriving Functor
```

would generate the following instance:

```
instance Functor Example where
  fmap f (Ex a1 a2 a3 a4) = Ex (f a1) a2 (fmap f a3) a4
```

The basic algorithm for *DeriveFunctor* (page 439) walks the arguments of each constructor of a data type, applying a mapping function depending on the type of each argument. If a plain type variable is found that is syntactically equivalent to the last type parameter of the data type (`a` in the above example), then we apply the function `f` directly to it. If a type is encountered that is not syntactically equivalent to the last type parameter *but does mention* the last type parameter somewhere in it, then a recursive call to `fmap` is made. If a type is found which doesn't mention the last type parameter at all, then it is left alone.

The second of those cases, in which a type is unequal to the type parameter but does contain the type parameter, can be surprisingly tricky. For example, the following example compiles:

```
newtype Right a = Right (Either Int a) deriving Functor
```

Modifying the code slightly, however, produces code which will not compile:

```
newtype Wrong a = Wrong (Either a Int) deriving Functor
```

The difference involves the placement of the last type parameter, `a`. In the `Right` case, `a` occurs within the type `Either Int a`, and moreover, it appears as the last type argument of `Either`. In the `Wrong` case, however, `a` is not the last type argument to `Either`; rather, `Int` is.

This distinction is important because of the way *DeriveFunctor* (page 439) works. The derived `Functor Right` instance would be:

```
instance Functor Right where
  fmap f (Right a) = Right (fmap f a)
```

Given a value of type `Right a`, GHC must produce a value of type `Right b`. Since the argument to the `Right` constructor has type `Either Int a`, the code recursively calls `fmap` on it to produce a value of type `Either Int b`, which is used in turn to construct a final value of type `Right b`.

The generated code for the `Functor Wrong` instance would look exactly the same, except with `Wrong` replacing every occurrence of `Right`. The problem is now that `fmap` is being applied recursively to a value of type `Either a Int`. This cannot possibly produce a value of type `Either b Int`, as `fmap` can only change the last type parameter! This causes the generated code to be ill-typed.

As a general rule, if a data type has a derived `Functor` instance and its last type parameter occurs on the right-hand side of the data declaration, then either it must (1) occur bare (e.g., `newtype Id a = Id a`), or (2) occur as the last argument of a type constructor (as in `Right` above).

There are two exceptions to this rule:

1. Tuple types. When a non-unit tuple is used on the right-hand side of a data declaration, *DeriveFunctor* (page 439) treats it as a product of distinct types. In other words, the following code:

   ```
   newtype Triple a = Triple (a, Int, [a]) deriving Functor
   ```

   Would result in a generated `Functor` instance like so:

   ```
   instance Functor Triple where
     fmap f (Triple a) =
       Triple (case a of
                    (a1, a2, a3) -> (f a1, a2, fmap f a3))
   ```

   That is, *DeriveFunctor* (page 439) pattern-matches its way into tuples and maps over each type that constitutes the tuple. The generated code is reminiscent of what would be generated from `data Triple a = Triple a Int [a]`, except with extra machinery to handle the tuple.

2. Function types. The last type parameter can appear anywhere in a function type as long as it occurs in a *covariant* position. To illustrate what this means, consider the following three examples:

   ```
   newtype CovFun1 a = CovFun1 (Int -> a) deriving Functor
   newtype CovFun2 a = CovFun2 ((a -> Int) -> a) deriving Functor
   newtype CovFun3 a = CovFun3 (((Int -> a) -> Int) -> a) deriving Functor
   ```

   All three of these examples would compile without issue. On the other hand:

   ```
   newtype ContraFun1 a = ContraFun1 (a -> Int) deriving Functor
   newtype ContraFun2 a = ContraFun2 ((Int -> a) -> Int) deriving Functor
   newtype ContraFun3 a = ContraFun3 (((a -> Int) -> a) -> Int) deriving Functor
   ```

   While these examples look similar, none of them would successfully compile. This is because all occurrences of the last type parameter `a` occur in *contravariant* positions, not covariant ones.

   Intuitively, a covariant type is *produced*, and a contravariant type is *consumed*. Most types in Haskell are covariant, but the function type is special in that the lefthand side of a function arrow reverses variance. If a function type `a -> b` appears in a covariant position (e.g., `CovFun1` above), then `a` is in a contravariant position and `b` is in a covariant position. Similarly, if `a -> b` appears in a contravariant position (e.g., `CovFun2` above), then `a` is in a covariant position and `b` is in a contravariant position.

   To see why a data type with a contravariant occurrence of its last type parameter cannot have a derived `Functor` instance, let's suppose that a `Functor ContraFun1` instance exists. The implementation would look something like this:

   ```
   instance Functor ContraFun1 where
     fmap f (ContraFun g) = ContraFun (\x -> _)
   ```

   We have `f :: a -> b`, `g :: a -> Int`, and `x :: b`. Using these, we must somehow fill in the hole (denoted with an underscore) with a value of type `Int`. What are our options?

We could try applying `g` to `x`. This won't work though, as `g` expects an argument of type `a`, and `x :: b`. Even worse, we can't turn `x` into something of type `a`, since `f` also needs an argument of type `a`! In short, there's no good way to make this work.

On the other hand, a derived `Functor` instances for the `CovFun`s are within the realm of possibility:

```
instance Functor CovFun1 where
  fmap f (CovFun1 g) = CovFun1 (\x -> f (g x))

instance Functor CovFun2 where
  fmap f (CovFun2 g) = CovFun2 (\h -> f (g (\x -> h (f x))))

instance Functor CovFun3 where
  fmap f (CovFun3 g) = CovFun3 (\h -> f (g (\k -> h (\x -> f (k x)))))
```

There are some other scenarios in which a derived `Functor` instance will fail to compile:

1. A data type has no type parameters (e.g., `data Nothing = Nothing`).
2. A data type's last type variable is used in a *`DatatypeContexts`* (page 322) constraint (e.g., `data Ord a => O a = O a`).
3. A data type's last type variable is used in an *`ExistentialQuantification`* (page 325) constraint, or is refined in a GADT. For example,

```
data T a b where
    T4 :: Ord b => b -> T a b
    T5 :: b -> T b b
    T6 :: T a (b,b)

deriving instance Functor (T a)
```

would not compile successfully due to the way in which `b` is constrained.

When the last type parameter has a phantom role (see *Roles* (page 417)), the derived `Functor` instance will not be produced using the usual algorithm. Instead, the entire value will be coerced.

```
data Phantom a = Z | S (Phantom a) deriving Functor
```

will produce the following instance:

```
instance Functor Phantom where
  fmap _ = coerce
```

When a type has no constructors, the derived `Functor` instance will simply force the (bottom) value of the argument using *`EmptyCase`* (page 300).

```
data V a deriving Functor
type role V nominal
```

will produce

```
instance Functor V where
  fmap _ z = case z of
```

6.6.4.2 Deriving Foldable instances

DeriveFoldable

Implied by *DeriveTraversable* (page 444)

Since 7.10.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow automatic deriving of instances for the `Foldable` typeclass.

With *DeriveFoldable* (page 442), one can derive `Foldable` instances for data types of kind `Type -> Type`. For example, this declaration:

```
data Example a = Ex a Char (Example a) (Example Char)
  deriving Foldable
```

would generate the following instance:

```
instance Foldable Example where
  foldr f z (Ex a1 a2 a3 a4) = f a1 (foldr f z a3)
  foldMap f (Ex a1 a2 a3 a4) = mappend (f a1) (foldMap f a3)
```

The algorithm for *DeriveFoldable* (page 442) is adapted from the *DeriveFunctor* (page 439) algorithm, but it generates definitions for `foldMap`, `foldr`, and `null` instead of `fmap`. In addition, *DeriveFoldable* (page 442) filters out all constructor arguments on the RHS expression whose types do not mention the last type parameter, since those arguments do not need to be folded over.

When the type parameter has a phantom role (see *Roles* (page 417)), *DeriveFoldable* (page 442) derives a trivial instance. For example, this declaration:

```
data Phantom a = Z | S (Phantom a)
```

will generate the following instance.

```
instance Foldable Phantom where
  foldMap _ _ = mempty
```

Similarly, when the type has no constructors, *DeriveFoldable* (page 442) will derive a trivial instance:

```
data V a deriving Foldable
type role V nominal
```

will generate the following.

```
instance Foldable V where
  foldMap _ _ = mempty
```

Here are the differences between the generated code for `Functor` and `Foldable`:

#. When a bare type variable `a` is encountered, *DeriveFunctor* (page 439) would generate `f a` for an `fmap` definition. *DeriveFoldable* (page 442) would generate `f a z` for `foldr`, `f a` for `foldMap`, and `False` for `null`.

1. When a type that is not syntactically equivalent to `a`, but which does contain `a`, is encountered, *DeriveFunctor* (page 439) recursively calls `fmap` on it. Similarly, *DeriveFoldable*

(page 442) would recursively call `foldr` and `foldMap`. Depending on the context, `null` may recursively call `null` or `all null`. For example, given

```
data F a = F (P a)
data G a = G (P (a, Int))
data H a = H (P (Q a))
```

`Foldable` deriving will produce

```
null (F x) = null x
null (G x) = null x
null (H x) = all null x
```

2. *DeriveFunctor* (page 439) puts everything back together again at the end by invoking the constructor. *DeriveFoldable* (page 442), however, builds up a value of some type. For `foldr`, this is accomplished by chaining applications of `f` and recursive `foldr` calls on the state value `z`. For `foldMap`, this happens by combining all values with `mappend`. For `null`, the values are usually combined with &&. However, if any of the values is known to be `False`, all the rest will be dropped. For example,

```
data SnocList a = Nil | Snoc (SnocList a) a
```

will not produce

```
null (Snoc xs _) = null xs && False
```

(which would walk the whole list), but rather

```
null (Snoc _ _) = False
```

There are some other differences regarding what data types can have derived `Foldable` instances:

1. Data types containing function types on the right-hand side cannot have derived `Foldable` instances.
2. `Foldable` instances can be derived for data types in which the last type parameter is existentially constrained or refined in a GADT. For example, this data type:

```
data E a where
    E1 :: (a ~ Int) => a   -> E a
    E2 ::              Int -> E Int
    E3 :: (a ~ Int) => a   -> E Int
    E4 :: (a ~ Int) => Int -> E a

deriving instance Foldable E
```

would have the following generated `Foldable` instance:

```
instance Foldable E where
    foldr f z (E1 e) = f e z
    foldr f z (E2 e) = z
    foldr f z (E3 e) = z
    foldr f z (E4 e) = z

    foldMap f (E1 e) = f e
    foldMap f (E2 e) = mempty
```

(continues on next page)

(continued from previous page)

```
    foldMap f (E3 e) = mempty
    foldMap f (E4 e) = mempty
```

Notice how every constructor of `E` utilizes some sort of existential quantification, but only the argument of `E1` is actually "folded over". This is because we make a deliberate choice to only fold over universally polymorphic types that are syntactically equivalent to the last type parameter. In particular:

- We don't fold over the arguments of `E1` or `E4` because even though `(a ~ Int)`, `Int` is not syntactically equivalent to `a`.
- We don't fold over the argument of `E3` because `a` is not universally polymorphic. The `a` in `E3` is (implicitly) existentially quantified, so it is not the same as the last type parameter of `E`.

6.6.4.3 Deriving `Traversable` instances

`DeriveTraversable`

Implies *`DeriveFoldable`* (page 442), *`DeriveFunctor`* (page 439)

Implied by *`GHC2021`* (page 271)

Since 7.10.1

Allow automatic deriving of instances for the `Traversable` typeclass.

With *`DeriveTraversable`* (page 444), one can derive `Traversable` instances for data types of kind `Type -> Type`. For example, this declaration:

```
data Example a = Ex a Char (Example a) (Example Char)
  deriving (Functor, Foldable, Traversable)
```

would generate the following `Traversable` instance:

```
instance Traversable Example where
  traverse f (Ex a1 a2 a3 a4)
    = fmap (\b1 b3 -> Ex b1 a2 b3 a4) (f a1) <*> traverse f a3
```

The algorithm for *`DeriveTraversable`* (page 444) is adapted from the *`DeriveFunctor`* (page 439) algorithm, but it generates a definition for `traverse` instead of `fmap`. In addition, *`DeriveTraversable`* (page 444) filters out all constructor arguments on the RHS expression whose types do not mention the last type parameter, since those arguments do not produce any effects in a traversal.

When the type parameter has a phantom role (see *Roles* (page 417)), *`DeriveTraversable`* (page 444) coerces its argument. For example, this declaration:

```
data Phantom a = Z | S (Phantom a) deriving Traversable
```

will generate the following instance:

```
instance Traversable Phantom where
  traverse _ z = pure (coerce z)
```

When the type has no constructors, *`DeriveTraversable`* (page 444) will derive the laziest instance it can.

```
data V a deriving Traversable
type role V nominal
```

will generate the following, using *EmptyCase* (page 300):

```
instance Traversable V where
  traverse _ z = pure (case z of)
```

Here are the differences between the generated code in each extension:

1. When a bare type variable `a` is encountered, both *DeriveFunctor* (page 439) and *DeriveTraversable* (page 444) would generate `f a` for an `fmap` and `traverse` definition, respectively.
2. When a type that is not syntactically equivalent to `a`, but which does contain `a`, is encountered, *DeriveFunctor* (page 439) recursively calls `fmap` on it. Similarly, *DeriveTraversable* (page 444) would recursively call `traverse`.
3. *DeriveFunctor* (page 439) puts everything back together again at the end by invoking the constructor. *DeriveTraversable* (page 444) does something similar, but it works in an `Applicative` context by chaining everything together with `(<*>)`.

Unlike *DeriveFunctor* (page 439), *DeriveTraversable* (page 444) cannot be used on data types containing a function type on the right-hand side.

For a full specification of the algorithms used in *DeriveFunctor* (page 439), *DeriveFoldable* (page 442), and *DeriveTraversable* (page 444), see this wiki page.

6.6.4.4 Deriving `Data` instances

`DeriveDataTypeable`

> **Since** 6.8.1
>
> **Status** Included in *GHC2024* (page 270), *GHC2021* (page 271)
>
> Enable automatic deriving of instances for the `Data` typeclass

6.6.4.5 Deriving `Typeable` instances

The class `Typeable` is very special:

- `Typeable` is kind-polymorphic (see *Kind polymorphism* (page 364)).
- GHC has a custom solver for discharging constraints that involve class `Typeable`, and handwritten instances are forbidden. This ensures that the programmer cannot subvert the type system by writing bogus instances.
- Derived instances of `Typeable` may be declared if the *DeriveDataTypeable* (page 445) extension is enabled, but they are ignored, and they may be reported as an error in a later version of the compiler.
- The rules for solving `Typeable` constraints are as follows:
  - A concrete type constructor applied to some types.

    ```
    instance (Typeable t1, .., Typeable t_n) =>
      Typeable (T t1 .. t_n)
    ```

This rule works for any concrete type constructor, including type constructors with polymorphic kinds. The only restriction is that if the type constructor has a polymorphic kind, then it has to be applied to all of its kinds parameters, and these kinds need to be concrete (i.e., they cannot mention kind variables).

- A type variable applied to some types:

```
instance (Typeable f, Typeable t1, .., Typeable t_n) =>
  Typeable (f t1 .. t_n)
```

- A concrete type literal.:

```
instance Typeable 0       -- Type natural literals
instance Typeable "Hello" -- Type-level symbols
```

6.6.4.6 Deriving `Lift` instances

DeriveLift

Since 8.0.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Enable automatic deriving of instances for the `Lift` typeclass for Template Haskell.

The class `Lift`, unlike other derivable classes, lives in `template-haskell` instead of `base`. Having a data type be an instance of `Lift` permits its values to be promoted to Template Haskell expressions (of type `ExpQ` and `Code Q a`), which can then be spliced into Haskell source code.

Here is an example of how one can derive `Lift`:

```
{-# LANGUAGE DeriveLift #-}
module Bar where

import Language.Haskell.TH.Syntax

data Foo a = Foo a | a :^: a deriving Lift

{-
instance (Lift a) => Lift (Foo a) where
    lift (Foo a) = [| Foo a |]
    lift ((:^:) u v) = [| (:^:) u v |]

    liftTyped (Foo a) = [|| Foo a ||]
    liftTyped ((:^:) u v) = [|| (:^:) u v ||]
-}

-----
{-# LANGUAGE TemplateHaskell #-}
module Baz where

import Bar
import Language.Haskell.TH.Lift

foo :: Foo String
foo = $(lift $ Foo "foo")

```

(continues on next page)

(continued from previous page)

```
fooExp :: Lift a => Foo a -> Q Exp
fooExp f = [| f |]
```

Note that the `Lift` typeclass takes advantage of *Representation polymorphism* (page 381) in order to support instances involving unboxed types. This means `DeriveLift` (page 446) also works for these types:

```
{-# LANGUAGE DeriveLift, MagicHash #-}
module Unboxed where

import GHC.Exts
import Language.Haskell.TH.Syntax

data IntHash = IntHash Int# deriving Lift

{-
instance Lift IntHash where
    lift (IntHash i) = [| IntHash i |]
    liftTyped (IntHash i) = [|| IntHash i ||]
-}
```

6.6.5 Generalised derived instances for newtypes

GeneralisedNewtypeDeriving
GeneralizedNewtypeDeriving

> **Since** 6.8.1. British spelling since 8.6.1.
>
> **Status** Included in `GHC2024` (page 270), `GHC2021` (page 271)

Enable GHC's cunning generalised deriving mechanism for `newtype`s

When you define an abstract type using `newtype`, you may want the new type to inherit some instances from its representation. In Haskell 98, you can inherit instances of `Eq`, `Ord`, `Enum` and `Bounded` by deriving them, but for any other classes you have to write an explicit instance declaration. For example, if you define

```
newtype Dollars = Dollars Int
```

and you want to use arithmetic on `Dollars`, you have to explicitly define an instance of `Num`:

```
instance Num Dollars where
  Dollars a + Dollars b = Dollars (a+b)
  ...
```

All the instance does is apply and remove the `newtype` constructor. It is particularly galling that, since the constructor doesn't appear at run-time, this instance declaration defines a dictionary which is *wholly equivalent* to the `Int` dictionary, only slower!

`DerivingVia` (page 457) (see *Deriving via* (page 457)) is a generalization of this idea.

6.6.5.1 Generalising the deriving clause

GHC now permits such instances to be derived instead, using the extension *GeneralizedNewtypeDeriving* (page 447), so one can write

```
newtype Dollars = Dollars { getDollars :: Int } deriving (Eq,Show,Num)
```

and the implementation uses the *same* Num dictionary for Dollars as for Int. In other words, GHC will generate something that resembles the following code

```
instance Num Int => Num Dollars
```

and then attempt to simplify the Num Int context as much as possible. GHC knows that there is a Num Int instance in scope, so it is able to discharge the Num Int constraint, leaving the code that GHC actually generates

```
instance Num Dollars
```

One can think of this instance being implemented with the same code as the Num Int instance, but with Dollars and getDollars added wherever necessary in order to make it typecheck. (In practice, GHC uses a somewhat different approach to code generation. See the *A more precise specification* (page 450) section below for more details.)

We can also derive instances of constructor classes in a similar way. For example, suppose we have implemented state and failure monad transformers, such that

```
instance Monad m => Monad (State s m)
instance Monad m => Monad (Failure m)
```

In Haskell 98, we can define a parsing monad by

```
type Parser tok m a = State [tok] (Failure m) a
```

which is automatically a monad thanks to the instance declarations above. With the extension, we can make the parser type abstract, without needing to write an instance of class Monad, via

```
newtype Parser tok m a = Parser (State [tok] (Failure m) a)
                         deriving Monad
```

In this case the derived instance declaration is of the form

```
instance Monad (State [tok] (Failure m)) => Monad (Parser tok m)
```

Notice that, since Monad is a constructor class, the instance is a *partial application* of the newtype, not the entire left hand side. We can imagine that the type declaration is "eta-converted" to generate the context of the instance declaration.

We can even derive instances of multi-parameter classes, provided the newtype is the last class parameter. In this case, a "partial application" of the class appears in the deriving clause. For example, given the class

```
class StateMonad s m | m -> s where ...
instance Monad m => StateMonad s (State s m) where ...
```

then we can derive an instance of StateMonad for Parser by

```
newtype Parser tok m a = Parser (State [tok] (Failure m) a)
                         deriving (Monad, StateMonad [tok])
```

The derived instance is obtained by completing the application of the class to the new type:

```
instance StateMonad [tok] (State [tok] (Failure m)) =>
         StateMonad [tok] (Parser tok m)
```

As a result of this extension, all derived instances in newtype declarations are treated uniformly (and implemented just by reusing the dictionary for the representation type), *except* Show and Read, which really behave differently for the newtype and its representation.

Note: It is sometimes necessary to enable additional language extensions when deriving instances via *GeneralizedNewtypeDeriving* (page 447). For instance, consider a simple class and instance using *UnboxedTuples* (page 556) syntax:

```
{-# LANGUAGE UnboxedTuples #-}

module Lib where

class AClass a where
  aMethod :: a -> (# Int, a #)

instance AClass Int where
  aMethod x = (# x, x #)
```

The following will fail with an "Illegal unboxed tuple" error, since the derived instance produced by the compiler makes use of unboxed tuple syntax,

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

import Lib

newtype Int' = Int' Int
             deriving (AClass)
```

However, enabling the *UnboxedTuples* (page 556) extension allows the module to compile. Similar errors may occur with a variety of extensions, including:

- *UnboxedTuples* (page 556)
- *PolyKinds* (page 364)
- *MultiParamTypeClasses* (page 470)
- *FlexibleContexts* (page 499)

6.6.5.2 A more precise specification

A derived instance is derived only for declarations of these forms (after expansion of any type synonyms)

```
newtype T v1..vn                   = MkT (t vk+1..vn) deriving (C t1..tj)
newtype instance T s1..sk vk+1..vn = MkT (t vk+1..vn) deriving (C t1..tj)
```

where

- `v1..vn` are type variables, and `t`, `s1..sk`, `t1..tj` are types.
- The `(C t1..tj)` is a partial applications of the class `C`, where the arity of `C` is exactly `j+1`. That is, `C` lacks exactly one type argument.
- `k` is chosen so that `C t1..tj (T v1...vk)` is well-kinded. (Or, in the case of a `data instance`, so that `C t1..tj (T s1..sk)` is well kinded.)
- The type `t` is an arbitrary type.
- The type variables `vk+1...vn` do not occur in the types `t`, `s1..sk`, or `t1..tj`.
- `C` is not `Read`, `Show`, `Typeable`, or `Data`. These classes should not "look through" the type or its constructor. You can still derive these classes for a newtype, but it happens in the usual way, not via this new mechanism. Confer with *Default deriving strategy* (page 456).
- It is safe to coerce each of the methods of `C`. That is, the missing last argument to `C` is not used at a nominal role in any of the `C`'s methods. (See *Roles* (page 417).)
- `C` is allowed to have associated type families, provided they meet the requirements laid out in the section on *GND and associated types* (page 451).

Then the derived instance declaration is of the form

```
instance C t1..tj t => C t1..tj (T v1...vk)
```

Note that if `C` does not contain any class methods, the instance context is wholly unnecessary, and as such GHC will instead generate:

```
instance C t1..tj (T v1..vk)
```

As an example which does *not* work, consider

```
newtype NonMonad m s = NonMonad (State s m s) deriving Monad
```

Here we cannot derive the instance

```
instance Monad (State s m) => Monad (NonMonad m)
```

because the type variable `s` occurs in `State s m`, and so cannot be "eta-converted" away. It is a good thing that this `deriving` clause is rejected, because `NonMonad m` is not, in fact, a monad — for the same reason. Try defining >>= with the correct type: you won't be able to.

Notice also that the *order* of class parameters becomes important, since we can only derive instances for the last one. If the `StateMonad` class above were instead defined as

```
class StateMonad m s | m -> s where ...
```

then we would not have been able to derive an instance for the `Parser` type above. We hypothesise that multi-parameter classes usually have one "main" parameter for which deriving new instances is most interesting.

Lastly, all of this applies only for classes other than `Read`, `Show`, `Typeable`, and `Data`, for which the stock derivation applies (section 4.3.3. of the Haskell Report). (For the standard classes `Eq`, `Ord`, `Ix`, and `Bounded` it is immaterial whether the stock method is used or the one described here.)

6.6.5.3 Associated type families

GeneralizedNewtypeDeriving (page 447) also works for some type classes with associated type families. Here is an example:

```
class HasRing a where
  type Ring a

newtype L1Norm a = L1Norm a
  deriving HasRing
```

The derived `HasRing` instance would look like

```
instance HasRing (L1Norm a) where
  type Ring (L1Norm a) = Ring a
```

To be precise, if the class being derived is of the form

```
class C c_1 c_2 ... c_m where
  type T1 t1_1 t1_2 ... t1_n
  ...
  type Tk tk_1 tk_2 ... tk_p
```

and the newtype is of the form

```
newtype N n_1 n_2 ... n_q = MkN <rep-type>
```

then you can derive a `C c_1 c_2 ... c_(m-1)` instance for `N n_1 n_2 ... n_q`, provided that:

- The type parameter `c_m` occurs once in each of the type variables of `T1` through `Tk`. Imagine a class where this condition didn't hold. For example:

  ```
  class Bad a b where
    type B a

  instance Bad Int a where
    type B Int = Char

  newtype Foo a = Foo a
    deriving (Bad Int)
  ```

  For the derived `Bad Int` instance, GHC would need to generate something like this:

  ```
  instance Bad Int (Foo a) where
    type B Int = B ???
  ```

Now we're stuck, since we have no way to refer to a on the right-hand side of the B family instance, so this instance doesn't really make sense in a *GeneralizedNewtypeDeriving* (page 447) setting.

- C does not have any associated data families (only type families). To see why data families are forbidden, imagine the following scenario:

```
class Ex a where
  data D a

instance Ex Int where
  data D Int = DInt Bool

newtype Age = MkAge Int deriving Ex
```

For the derived Ex instance, GHC would need to generate something like this:

```
instance Ex Age where
  data D Age = ???
```

But it is not clear what GHC would fill in for ???, as each data family instance must generate fresh data constructors.

If both of these conditions are met, GHC will generate this instance:

```
instance C c_1 c_2 ... c_(m-1) <rep-type> =>
         C c_1 c_2 ... c_(m-1) (N n_1 n_2 ... n_q) where
  type T1 t1_1 t1_2 ... (N n_1 n_2 ... n_q) ... t1_n
     = T1 t1_1 t1_2 ... <rep-type>          ... t1_n
  ...
  type Tk tk_1 tk_2 ... (N n_1 n_2 ... n_q) ... tk_p
     = Tk tk_1 tk_2 ... <rep-type>          ... tk_p
```

Again, if C contains no class methods, the instance context will be redundant, so GHC will instead generate `instance C c_1 c_2 ... c_(m-1) (N n_1 n_2 ... n_q)`.

Beware that in some cases, you may need to enable the *UndecidableInstances* (page 484) extension in order to use this feature. Here's a pathological case that illustrates why this might happen:

```
class C a where
  type T a

newtype Loop = MkLoop Loop
  deriving C
```

This will generate the derived instance:

```
instance C Loop where
  type T Loop = T Loop
```

Here, it is evident that attempting to use the type T Loop will throw the typechecker into an infinite loop, as its definition recurses endlessly. In other cases, you might need to enable *UndecidableInstances* (page 484) even if the generated code won't put the typechecker into a loop. For example:

```
instance C Int where
  type C Int = Int
```

(continues on next page)

(continued from previous page)

```
newtype MyInt = MyInt Int
  deriving C
```

This will generate the derived instance:

```
instance C MyInt where
  type T MyInt = T Int
```

Although typechecking `T MyInt` will terminate, GHC's termination checker isn't sophisticated enough to determine this, so you'll need to enable *UndecidableInstances* (page 484) in order to use this derived instance. If you do go down this route, make sure you can convince yourself that all of the type family instances you're deriving will eventually terminate if used!

Note that *DerivingVia* (page 457) (see *Deriving via* (page 457)) uses essentially the same specification to derive instances of associated type families as well (except that it uses the `via` type instead of the underlying `rep-type` of a newtype).

6.6.6 Deriving any other class

DeriveAnyClass

Since 7.10.1

Allow use of any typeclass in `deriving` clauses.

With *DeriveAnyClass* (page 453) you can derive any other class. The compiler will simply generate an instance declaration with no explicitly-defined methods. This is mostly useful in classes whose *minimal set* (page **??**) is empty, and especially when writing *generic functions* (page **??**).

As an example, consider a simple pretty-printer class `SPretty`, which outputs pretty strings:

```
{-# LANGUAGE DefaultSignatures, DeriveAnyClass #-}

class SPretty a where
  sPpr :: a -> String
  default sPpr :: Show a => a -> String
  sPpr = show
```

If a user does not provide a manual implementation for `sPpr`, then it will default to `show`. Now we can leverage the *DeriveAnyClass* (page 453) extension to easily implement a `SPretty` instance for a new data type:

```
data Foo = Foo deriving (Show, SPretty)
```

The above code is equivalent to:

```
data Foo = Foo deriving Show
instance SPretty Foo
```

That is, an `SPretty Foo` instance will be created with empty implementations for all methods. Since we are using *DefaultSignatures* (page 472) in this example, a default implementation of `sPpr` is filled in automatically.

Note the following details

- In case you try to derive some class on a newtype, and *GeneralizedNewtypeDeriving* (page 447) is also on, *DeriveAnyClass* (page 453) takes precedence.
- The instance context is determined by the type signatures of the derived class's methods. For instance, if the class is:

```
class Foo a where
  bar :: a -> String
  default bar :: Show a => a -> String
  bar = show

  baz :: a -> a -> Bool
  default baz :: Ord a => a -> a -> Bool
  baz x y = compare x y == EQ
```

  And you attempt to derive it using *DeriveAnyClass* (page 453):

```
instance Eq   a => Eq   (Option a) where ...
instance Ord  a => Ord  (Option a) where ...
instance Show a => Show (Option a) where ...

data Option a = None | Some a deriving Foo
```

  Then the derived Foo instance will be:

```
instance (Show a, Ord a) => Foo (Option a)
```

  Since the default type signatures for bar and baz require Show a and Ord a constraints, respectively.

  Constraints on the non-default type signatures can play a role in inferring the instance context as well. For example, if you have this class:

```
class HigherEq f where
  (==#) :: f a -> f a -> Bool
  default (==#) :: Eq (f a) => f a -> f a -> Bool
  x ==# y = (x == y)
```

  And you tried to derive an instance for it:

```
instance Eq a => Eq (Option a) where ...
data Option a = None | Some a deriving HigherEq
```

  Then it will fail with an error to the effect of:

```
No instance for (Eq a)
    arising from the 'deriving' clause of a data type declaration
```

  That is because we require an Eq (Option a) instance from the default type signature for (==#), which in turn requires an Eq a instance, which we don't have in scope. But if you tweak the definition of HigherEq slightly:

```
class HigherEq f where
  (==#) :: Eq a => f a -> f a -> Bool
  default (==#) :: Eq (f a) => f a -> f a -> Bool
  x ==# y = (x == y)
```

  Then it becomes possible to derive a HigherEq Option instance. Note that the only difference is that now the non-default type signature for (==#) brings in an Eq a constraint.

Constraints from non-default type signatures never appear in the derived instance context itself, but they can be used to discharge obligations that are demanded by the default type signatures. In the example above, the default type signature demanded an `Eq a` instance, and the non-default signature was able to satisfy that request, so the derived instance is simply:

```
instance HigherEq Option
```

- *DeriveAnyClass* (page 453) can be used with partially applied classes, such as

  ```
  data T a = MKT a deriving( D Int )
  ```

  which generates

  ```
  instance D Int a => D Int (T a) where {}
  ```

- *DeriveAnyClass* (page 453) can be used to fill in default instances for associated type families:

  ```
  {-# LANGUAGE DeriveAnyClass, TypeFamilies #-}

  class Sizable a where
    type Size a
    type Size a = Int

  data Bar = Bar deriving Sizable

  doubleBarSize :: Size Bar -> Size Bar
  doubleBarSize s = 2*s
  ```

  The `deriving( Sizable )` is equivalent to saying

  ```
  instance Sizeable Bar where {}
  ```

  and then the normal rules for filling in associated types from the default will apply, making `Size Bar` equal to `Int`.

6.6.7 Deriving strategies

DerivingStrategies

Since 8.2.1

Status Included in *GHC2024* (page 270)

Allow multiple `deriving`, each optionally qualified with a *strategy*.

In most scenarios, every `deriving` statement generates a typeclass instance in an unambiguous fashion. There is a corner case, however, where simultaneously enabling both the *GeneralizedNewtypeDeriving* (page 447) and *DeriveAnyClass* (page 453) extensions can make deriving become ambiguous. Consider the following example

```
{-# LANGUAGE DeriveAnyClass, GeneralizedNewtypeDeriving #-}
newtype Foo = MkFoo Bar deriving C
```

One could either pick the `DeriveAnyClass` approach to deriving `C` or the `GeneralizedNewtypeDeriving` approach to deriving `C`, both of which would be equally as valid. GHC defaults to favoring `DeriveAnyClass` in such a dispute, but this is not a satisfying solution, since that leaves users unable to use both language extensions in a single module.

To make this more robust, GHC has a notion of deriving strategies, which allow the user to explicitly request which approach to use when deriving an instance. To enable this feature, one must enable the *DerivingStrategies* (page 455) language extension. A deriving strategy can be specified in a deriving clause

```
newtype Foo = MkFoo Bar
  deriving newtype C
```

Or in a standalone deriving declaration

```
deriving anyclass instance C Foo
```

DerivingStrategies (page 455) also allows the use of multiple deriving clauses per data declaration so that a user can derive some instance with one deriving strategy and other instances with another deriving strategy. For example

```
newtype Baz = Baz Quux
  deriving          (Eq, Ord)
  deriving stock    (Read, Show)
  deriving newtype  (Num, Floating)
  deriving anyclass C
```

Currently, the deriving strategies are:

- `stock`: Have GHC implement a "standard" instance for a data type, if possible (e.g., `Eq`, `Ord`, `Generic`, `Data`, `Functor`, etc.)
- `anyclass`: Use *DeriveAnyClass* (page 453) (see *Deriving any other class* (page 453))
- **`newtype`: Use *GeneralizedNewtypeDeriving* (page 447)** (see *Generalised derived instances for newtypes* (page 447))
- `via`: Use *DerivingVia* (page 457) (see *Deriving via* (page 457))

6.6.7.1 Default deriving strategy

If an explicit deriving strategy is not given, multiple strategies may apply. In that case, GHC chooses the strategy as follows:

1. Stock type classes, i.e. those specified in the report and those enabled by *language extensions* (page 438), are derived using the `stock` strategy, with the following exception:
   - For newtypes, `Eq`, `Ord`, `Ix` and `Bounded` are always derived using the `newtype` strategy, even without `GeneralizedNewtypeDeriving` enabled. (There should be no observable difference to instances derived using the stock strategy.)
   - Also for newtypes, `Functor`, `Foldable` and `Enum` are derived using the `newtype` strategy if `GeneralizedNewtypeDeriving` is enabled and the derivation succeeds.
2. For any other type class:
   1. When *DeriveAnyClass* (page 453) is enabled, use `anyclass`.
   2. When *GeneralizedNewtypeDeriving* (page 447) is enabled and we are deriving for a newtype, then use `newtype`.

If both rules apply to a deriving clause, then `anyclass` is used and the user is warned about the ambiguity. The warning can be avoided by explicitly stating the desired deriving strategy.

6.6.8 Deriving via

DerivingVia

Implies *DerivingStrategies* (page 455)

Since 8.6.1

This allows `deriving` a class instance for a type by specifying another type that is already an instance of that class. This only makes sense if the methods have identical runtime representations, in the sense that coerce (see The `Coercible` constraint) can convert the existing implementation into the desired implementation. The generated code will be rejected with a type error otherwise.

DerivingVia (page 457) is indicated by the use of the `via` deriving strategy. `via` requires specifying another type (the `via` type) to `coerce` through. For example, this code:

```
{-# LANGUAGE DerivingVia #-}

import Numeric

newtype Hex a = Hex a

instance (Integral a, Show a) => Show (Hex a) where
  show (Hex a) = "0x" ++ showHex a ""

newtype Unicode = U Int
  deriving Show
    via (Hex Int)

-- >>> euroSign
-- 0x20ac
euroSign :: Unicode
euroSign = U 0x20ac
```

Generates the following instance

```
instance Show Unicode where
  show :: Unicode -> String
  show = Data.Coerce.coerce
    @(Hex Int -> String)
    @(Unicode -> String)
    show
```

This extension generalizes *GeneralizedNewtypeDeriving* (page 447). To derive `Num Unicode` with GND (`deriving newtype Num`) it must reuse the `Num Int` instance. With `DerivingVia`, we can explicitly specify the representation type `Int`:

```
newtype Unicode = U Int
  deriving Num
    via Int

  deriving Show
```

(continues on next page)

(continued from previous page)

```
    via (Hex Int)

euroSign :: Unicode
euroSign = 0x20ac
```

Code duplication is common in instance declarations. A familiar pattern is lifting operations over an `Applicative` functor. Instead of having catch-all instances for `f a` which overlap with all other such instances, like so:

```
instance (Applicative f, Semigroup a) => Semigroup (f a) ..
instance (Applicative f, Monoid    a) => Monoid    (f a) ..
```

We can instead create a newtype `App` (where `App f a` and `f a` are represented the same in memory) and use *DerivingVia* (page 457) to explicitly enable uses of this pattern:

```
{-# LANGUAGE DerivingVia, DeriveFunctor, GeneralizedNewtypeDeriving #-}

import Control.Applicative

newtype App f a = App (f a) deriving newtype (Functor, Applicative)

instance (Applicative f, Semigroup a) => Semigroup (App f a) where
  (<>) = liftA2 (<>)

instance (Applicative f, Monoid a) => Monoid (App f a) where
  mempty = pure mempty

data Pair a = MkPair a a
  deriving stock
    Functor

  deriving (Semigroup, Monoid)
    via (App Pair a)

instance Applicative Pair where
  pure a = MkPair a a

  MkPair f g <*> MkPair a b = MkPair (f a) (g b)
```

Note that the `via` type does not have to be a `newtype`. The only restriction is that it is coercible with the original data type. This means there can be arbitrary nesting of newtypes, as in the following example:

```
newtype Kleisli m a b = Kleisli (a -> m b)
  deriving (Semigroup, Monoid)
    via (a -> App m b)
```

Here we make use of the `Monoid ((->) a)` instance.

When used in combination with *StandaloneDeriving* (page 436) we swap the order for the instance we base our derivation on and the instance we define e.g.:

```
deriving via (a -> App m b) instance Monoid (Kleisli m a b)
```

6.7 Patterns

6.7.1 Pattern guards

NoPatternGuards

Since 6.8.1

Status Included in *Haskell98* (page 272)

Disable pattern guards.

6.7.2 View patterns

ViewPatterns

Since 6.10.1

Allow use of view pattern syntax.

View patterns are enabled by the language extension *ViewPatterns* (page 459). More information and examples of view patterns can be found on the Wiki page.

View patterns are somewhat like pattern guards that can be nested inside of other patterns. They are a convenient way of pattern-matching against values of abstract types. For example, in a programming language implementation, we might represent the syntax of the types of the language as follows:

```
type Typ

data TypView = Unit
             | Arrow Typ Typ

view :: Typ -> TypView

-- additional operations for constructing Typ's ...
```

The representation of Typ is held abstract, permitting implementations to use a fancy representation (e.g., hash-consing to manage sharing). Without view patterns, using this signature is a little inconvenient:

```
size :: Typ -> Integer
size t = case view t of
  Unit -> 1
  Arrow t1 t2 -> size t1 + size t2
```

It is necessary to iterate the case, rather than using an equational function definition. And the situation is even worse when the matching against `t` is buried deep inside another pattern.

View patterns permit calling the view function inside the pattern and matching against the result:

```
size (view -> Unit) = 1
size (view -> Arrow t1 t2) = size t1 + size t2
```

That is, we add a new form of pattern, written ⟨expression⟩ `->` ⟨pattern⟩ that means “apply the expression to whatever we’re trying to match against, and then match the result of that application against the pattern". The expression can be any Haskell expression of function type, and view patterns can be used wherever patterns are used.

The semantics of a pattern (⟨exp⟩ -> ⟨pat⟩) are as follows:

- Scoping: The variables bound by the view pattern are the variables bound by ⟨pat⟩.

  Any variables in ⟨exp⟩ are bound occurrences, but variables bound "to the left" in a pattern are in scope. This feature permits, for example, one argument to a function to be used in the view of another argument. For example, the function `clunky` from *Pattern guards* (page 459) can be written using view patterns as follows:

  ```
  clunky env (lookup env -> Just val1) (lookup env -> Just val2) = val1 + val2
  ...other equations for clunky...
  ```

  More precisely, the scoping rules are:

  - In a single pattern, variables bound by patterns to the left of a view pattern expression are in scope. For example:

    ```
    example :: Maybe ((String -> Integer,Integer), String) -> Bool
    example (Just ((f,_), f -> 4)) = True
    ```

    Additionally, in function definitions, variables bound by matching earlier curried arguments may be used in view pattern expressions in later arguments:

    ```
    example :: (String -> Integer) -> String -> Bool
    example f (f -> 4) = True
    ```

    That is, the scoping is the same as it would be if the curried arguments were collected into a tuple.

  - In mutually recursive bindings, such as `let`, `where`, or the top level, view patterns in one declaration may not mention variables bound by other declarations. That is, each declaration must be self-contained. For example, the following program is not allowed:

    ```
    let {(x -> y) = e1 ;
         (y -> x) = e2 } in x
    ```

  (For some amplification on this design choice see #4061.

- Typing: If ⟨exp⟩ has type ⟨T1⟩ -> ⟨T2⟩ and ⟨pat⟩ matches a ⟨T2⟩, then the whole view pattern matches a ⟨T1⟩.

- Matching: To the equations in Section 3.17.3 of the Haskell 98 Report, add the following:

  ```
  case v of { (e -> p) -> e1 ; _ -> e2 }
   =
  case (e v) of { p -> e1 ; _ -> e2 }
  ```

  That is, to match a variable ⟨v⟩ against a pattern (⟨exp⟩ -> ⟨pat⟩), evaluate (⟨exp⟩ ⟨v⟩) and match the result against ⟨pat⟩.

- Efficiency: When the same view function is applied in multiple branches of a function definition or a case expression (e.g., in `size` above), GHC makes an attempt to collect these applications into a single nested case expression, so that the view function is only applied once. Pattern compilation in GHC follows the matrix algorithm described in Chapter 4 of The Implementation of Functional Programming Languages. When the top rows of the first column of a matrix are all view patterns with the "same" expression, these patterns are transformed into a single nested case. This includes, for example, adjacent view patterns that line up in a tuple, as in

```
f ((view -> A, p1), p2) = e1
f ((view -> B, p3), p4) = e2
```

The current notion of when two view pattern expressions are "the same" is very restricted: it is not even full syntactic equality. However, it does include variables, literals, applications, and tuples; e.g., two instances of `view ("hi", "there")` will be collected. However, the current implementation does not compare up to alpha-equivalence, so two instances of `(x, view x -> y)` will not be coalesced.

6.7.3 n+k patterns

NPlusKPatterns

Since 6.12.1

Status Included in *Haskell98* (page 272)

Enable use of `n+k` patterns.

6.7.4 Pattern synonyms

PatternSynonyms

Since 7.8.1

Allow the definition of pattern synonyms.

Pattern synonyms are enabled by the language extension *PatternSynonyms* (page 461), which is required for defining them, but *not* for using them. More information and examples of pattern synonyms can be found on the Wiki page.

Pattern synonyms enable giving names to parametrized pattern schemes. They can also be thought of as abstract constructors that don't have a bearing on data representation. For example, in a programming language implementation, we might represent types of the language as follows:

```
data Type = App String [Type]
```

Here are some examples of using said representation. Consider a few types of the `Type` universe encoded like this:

```
App "->" [t1, t2]          -- t1 -> t2
App "Int" []               -- Int
App "Maybe" [App "Int" []] -- Maybe Int
```

This representation is very generic in that no types are given special treatment. However, some functions might need to handle some known types specially, for example the following two functions collect all argument types of (nested) arrow types, and recognize the `Int` type, respectively:

```
collectArgs :: Type -> [Type]
collectArgs (App "->" [t1, t2]) = t1 : collectArgs t2
collectArgs _                   = []
```

(continues on next page)

(continued from previous page)

```
isInt :: Type -> Bool
isInt (App "Int" []) = True
isInt _              = False
```

Matching on App directly is both hard to read and error prone to write. And the situation is even worse when the matching is nested:

```
isIntEndo :: Type -> Bool
isIntEndo (App "->" [App "Int" [], App "Int" []]) = True
isIntEndo _                                        = False
```

Pattern synonyms permit abstracting from the representation to expose matchers that behave in a constructor-like manner with respect to pattern matching. We can create pattern synonyms for the known types we care about, without committing the representation to them (note that these don't have to be defined in the same module as the Type type):

```
pattern Arrow t1 t2 = App "->"    [t1, t2]
pattern Int         = App "Int"   []
pattern Maybe t     = App "Maybe" [t]
```

Which enables us to rewrite our functions in a much cleaner style:

```
collectArgs :: Type -> [Type]
collectArgs (Arrow t1 t2) = t1 : collectArgs t2
collectArgs _             = []

isInt :: Type -> Bool
isInt Int = True
isInt _   = False

isIntEndo :: Type -> Bool
isIntEndo (Arrow Int Int) = True
isIntEndo _               = False
```

In general there are three kinds of pattern synonyms. Unidirectional, bidirectional and explicitly bidirectional. The examples given so far are examples of bidirectional pattern synonyms. A bidirectional synonym behaves the same as an ordinary data constructor. We can use it in a pattern context to deconstruct values and in an expression context to construct values. For example, we can construct the value *intEndo* using the pattern synonyms *Arrow* and *Int* as defined previously.

```
intEndo :: Type
intEndo = Arrow Int Int
```

This example is equivalent to the much more complicated construction if we had directly used the *Type* constructors.

```
intEndo :: Type
intEndo = App "->" [App "Int" [], App "Int" []]
```

Unidirectional synonyms can only be used in a pattern context and are defined as follows:

```
pattern Head x <- x:xs
```

In this case, Head ⟨x⟩ cannot be used in expressions, only patterns, since it wouldn't specify a value for the ⟨xs⟩ on the right-hand side. However, we can define an explicitly bidirectional pattern synonym by separately specifying how to construct and deconstruct a type. The syntax for doing this is as follows:

```
pattern HeadC x <- x:xs where
  HeadC x = [x]
```

We can then use HeadC in both expression and pattern contexts. In a pattern context it will match the head of any list with length at least one. In an expression context it will construct a singleton list.

Explicitly bidirectional pattern synonyms offer greater flexibility than implicitly bidirectional ones in terms of the syntax that is permitted. For instance, the following is not a legal implicitly bidirectional pattern synonym:

```
pattern StrictJust a = Just !a
```

This is illegal because the use of *BangPatterns* (page 541) on the right-hand sides prevents it from being a well formed expression. However, constructing a strict pattern synonym is quite possible with an explicitly bidirectional pattern synonym:

```
pattern StrictJust a <- Just !a where
  StrictJust !a = Just a
```

Constructing an explicitly bidirectional pattern synonym also:

- can create different data constructors from the underlying data type, not just the one appearing in the pattern match;
- can call any functions or conditional logic, especially validation, of course providing it constructs a result of the right type;
- can use guards on the lhs of the =;
- can have multiple equations.

For example:

```
data PosNeg = Pos Int | Neg Int
pattern Smarter{ nonneg } <- Pos nonneg  where
  Smarter x = if x >= 0 then (Pos x) else (Neg x)
```

Or using guards:

```
pattern Smarter{ nonneg } <- Pos nonneg  where
  Smarter x | x >= 0    = (Pos x)
            | otherwise = (Neg x)
```

There is an extensive Haskell folk art of smart constructors, essentially functions that wrap validation around a constructor, and avoid exposing its representation. The downside is that the underlying constructor can't be used as a matcher. Pattern synonyms can be used as genuinely smart constructors, for both validation and matching.

The table below summarises where each kind of pattern synonym can be used.

Context	Unidirectional	Bidirectional	Explicitly Bidirectional
Pattern	Yes	Yes	Yes
Expression	No	Yes (Inferred)	Yes (Explicit)

6.7.4.1 Record Pattern Synonyms

It is also possible to define pattern synonyms which behave just like record constructors. The syntax for doing this is as follows:

```
pattern Point :: Int -> Int -> (Int, Int)
pattern Point{x, y} = (x, y)
```

The idea is that we can then use `Point` just as if we had defined a new datatype `MyPoint` with two fields `x` and `y`.

```
data MyPoint = Point { x :: Int, y :: Int }
```

Whilst a normal pattern synonym can be used in two ways, there are then seven ways in which to use `Point`. Precisely the ways in which a normal record constructor can be used.

Usage	Example
As a constructor	`zero = Point 0 0`
As a constructor with record syntax	`zero = Point { x = 0, y = 0 }`
In a pattern context	`isZero (Point 0 0) = True`
In a pattern context with record syntax	`isZero (Point { x = 0, y = 0 }) = True`
In a pattern context with field puns	`getX (Point {x}) = x`
In a record update	`(0, 0) { x = 1 } == (1,0)`
Using record selectors	`x (0,0) == 0`

For a unidirectional record pattern synonym we define record selectors but do not allow record updates or construction.

The syntax and semantics of pattern synonyms are elaborated in the following subsections. There are also lots more details in the paper.

See the Wiki page for more details.

6.7.4.2 Syntax and scoping of pattern synonyms

A pattern synonym declaration can be either unidirectional, bidirectional or explicitly bidirectional. The syntax for unidirectional pattern synonyms is:

```
pattern pat_lhs <- pat
```

the syntax for bidirectional pattern synonyms is:

```
pattern pat_lhs = pat
```

and the syntax for explicitly bidirectional pattern synonyms is:

```
pattern pat_lhs <- pat where
  pat_lhs = expr                      -- lhs restricted, see below
```

We can define either prefix, infix or record pattern synonyms by modifying the form of *pat_lhs*. The syntax for these is as follows:

Prefix	`Name args`
Infix	``arg1 `Name` arg2`` or `arg1 op arg2`
Record	`Name{arg1,arg2,...,argn}`

The *pat_lhs* for explicitly bidirectional construction cannot use Record syntax. (Because the rhs *expr* might be constructing different data constructors.) It can use guards with multiple equations.

Pattern synonym declarations can only occur in the top level of a module. In particular, they are not allowed as local definitions.

The variables in the left-hand side of the definition are bound by the pattern on the right-hand side. For bidirectional pattern synonyms, all the variables of the right-hand side must also occur on the left-hand side; also, wildcard patterns and view patterns are not allowed. For unidirectional and explicitly bidirectional pattern synonyms, there is no restriction on the right-hand side pattern.

Pattern synonyms cannot be defined recursively.

COMPLETE pragmas (page 619) can be specified in order to tell the pattern match exhaustiveness checker that a set of pattern synonyms is complete.

6.7.4.3 Import and export of pattern synonyms

The name of the pattern synonym is in the same namespace as proper data constructors. Like normal data constructors, pattern synonyms can be imported and exported through association with a type constructor or independently.

To export them on their own, in an export or import specification, you must prefix pattern names with the `pattern` keyword, e.g.:

```
module Example (pattern Zero) where

data MyNum = MkNum Int

pattern Zero :: MyNum
pattern Zero = MkNum 0
```

Without the `pattern` prefix, `Zero` would be interpreted as a type constructor in the export list.

You may also use the `pattern` keyword in an import/export specification to import or export an ordinary data constructor. For example:

```
import Data.Maybe( pattern Just )
```

would bring into scope the data constructor `Just` from the `Maybe` type, without also bringing the type constructor `Maybe` into scope.

As of GHC 8.0.1 you may also "bundle" pattern synonyms with an exported type constructor, making that pattern appear as a data constructor of that type. To bundle a pattern synonym, we list the pattern synonym in the export list of a module which exports the type constructor. For example, to bundle `Zero` with `MyNum` we could write the following:

```
module Example ( MyNum(Zero) ) where
```

If a module was then to import `MyNum` from `Example`, it would also import the pattern synonym `Zero`.

It is also possible to use the special token `..` in an export list to mean all currently bundled constructors. For example, we could write:

```
module Example ( MyNum(.., Zero) ) where
```

in which case, `Example` would export the type constructor `MyNum` with the data constructor `MkNum` and also the pattern synonym `Zero`.

Bundled pattern synonyms are type checked to ensure that they are of the same type as the type constructor which they are bundled with. A pattern synonym `P` can not be bundled with a type constructor `T` if `P`'s type is visibly incompatible with `T`.

A module which imports `MyNum(..)` from `Example` and then re-exports `MyNum(..)` will also export any pattern synonyms bundled with `MyNum` in `Example`. A more complete specification can be found on the wiki.

6.7.4.4 Typing of pattern synonyms

Given a pattern synonym definition of the form

```
pattern P var1 var2 ... varN <- pat
```

it is assigned a *pattern type* of the form

```
pattern P :: CReq => CProv => t1 -> t2 -> ... -> tN -> t
```

where ⟨CReq⟩ and ⟨CProv⟩ are type contexts, and ⟨t1⟩, ⟨t2⟩, ..., ⟨tN⟩ and ⟨t⟩ are types. Notice the unusual form of the type, with two contexts ⟨CReq⟩ and ⟨CProv⟩:

- ⟨CReq⟩ are the constraints *required* to match the pattern.
- ⟨CProv⟩ are the constraints *made available (provided)* by a successful pattern match.

For example, consider

```
data T a where
  MkT :: (Show b) => a -> b -> T a

f1 :: (Num a, Eq a) => T a -> String
f1 (MkT 42 x) = show x

pattern ExNumPat :: (Num a, Eq a) => (Show b) => b -> T a
pattern ExNumPat x = MkT 42 x

f2 :: (Eq a, Num a) => T a -> String
f2 (ExNumPat x) = show x
```

Here `f1` does not use pattern synonyms. To match against the numeric pattern 42 *requires* the caller to satisfy the constraints `(Num a, Eq a)`, so they appear in `f1`'s type. The call to `show` generates a `(Show b)` constraint, where `b` is an existentially type variable bound by the pattern match on `MkT`. But the same pattern match also *provides* the constraint `(Show b)` (see `MkT`'s type), and so all is well.

Exactly the same reasoning applies to `ExNumPat`: matching against `ExNumPat` *requires* the constraints `(Num a, Eq a)`, and *provides* the constraint `(Show b)`.

Note also the following points

- In the common case where `CProv` is empty, (i.e., `()`), it can be omitted altogether in the above pattern type signature for `P`.
- However, if `CProv` is non-empty, while `CReq` is, the above pattern type signature for `P` must be specified as

```
P :: () => CProv => t1 -> t2 -> .. -> tN -> t
```

- The GHCi *:info* (page 48) command shows pattern types in this format.
- You may specify an explicit *pattern signature*, as we did for ExNumPat above, to specify the type of a pattern, just as you can for a function. As usual, the type signature can be less polymorphic than the inferred type. For example

```
-- Inferred type would be 'a -> [a]'
pattern SinglePair :: (a, a) -> [(a, a)]
pattern SinglePair x = [x]
```

  Just like signatures on value-level bindings, pattern synonym signatures can apply to more than one pattern. For instance,

```
pattern Left', Right' :: a -> Either a a
pattern Left' x  = Left x
pattern Right' x = Right x
```

- The rules for lexically-scoped type variables (see *Lexically scoped type variables* (page 512)) apply to pattern-synonym signatures. As those rules specify, only the type variables from an explicit, syntactically-visible outer *forall* (the universals) scope over the definition of the pattern synonym; the existentials, bound by the inner forall, do not. For example

```
data T a where
   MkT :: Bool -> b -> (b->Int) -> a -> T a

pattern P :: forall a. forall b. b -> (b->Int) -> a -> T a
pattern P x y v <- MkT True x y (v::a)
```

  Here the universal type variable *a* scopes over the definition of *P*, but the existential *b* does not. (c.f. discussion on #14998.)

- For a bidirectional pattern synonym, a use of the pattern synonym as an expression has the type

```
(CReq, CProv) => t1 -> t2 -> ... -> tN -> t
```

  So in the previous example, when used in an expression, ExNumPat has type

```
ExNumPat :: (Num a, Eq a, Show b) => b -> T t
```

  Notice that this is a tiny bit more restrictive than the expression MkT 42 x which would not require (Eq a).

- Consider these two pattern synonyms:

```
data S a where
   S1 :: Bool -> S Bool

pattern P1 :: Bool -> Maybe Bool
pattern P1 b = Just b

pattern P2 :: () => (b ~ Bool) => Bool -> S b
pattern P2 b = S1 b
```

(continues on next page)

(continued from previous page)

```
f :: Maybe a -> String
f (P1 x) = "no no no"     -- Type-incorrect

g :: S a -> String
g (P2 b) = "yes yes yes"  -- Fine
```

Pattern P1 can only match against a value of type Maybe Bool, so function f is rejected because the type signature is Maybe a. (To see this, imagine expanding the pattern synonym.)

On the other hand, function g works fine, because matching against P2 (which wraps the GADT S) provides the local equality (a~Bool). If you were to give an explicit pattern signature P2 :: Bool -> S Bool, then P2 would become less polymorphic, and would behave exactly like P1 so that g would then be rejected.

In short, if you want GADT-like behaviour for pattern synonyms, then (unlike concrete data constructors like S1) you must write its type with explicit provided equalities. For a concrete data constructor like S1 you can write its type signature as either S1 :: Bool -> S Bool or S1 :: (b~Bool) => Bool -> S b; the two are equivalent. Not so for pattern synonyms: the two forms are different, in order to distinguish the two cases above. (See #9953 for discussion of this choice.)

6.7.4.5 Matching of pattern synonyms

A pattern synonym occurrence in a pattern is evaluated by first matching against the pattern synonym itself, and then on the argument patterns.

More precisely, the semantics of pattern matching is given in Section 3.17 of the Haskell 2010 report. To the informal semantics in Section 3.17.2 we add this extra rule:

- If the pattern is a constructor pattern (P p1 ... pn), where P is a pattern synonym defined by P x1 ... xn = p or P x1 ... xn <- p, then:
  (a) Match the value v against p. If this match fails or diverges, so does the whole (pattern synonym) match. Otherwise the match against p must bind the variables x1 ... xn; let them be bound to values v1 ... vn.
  (b) Match v1 against p1, v2 against p2 and so on. If any of these matches fail or diverge, so does the whole match.
  (c) If all the matches against the pi succeed, the match succeeds, binding the variables bound by the pi . (The xi are not bound; they remain local to the pattern synonym declaration.)

For example, in the following program, f and f' are equivalent:

```
pattern Pair x y <- [x, y]

f (Pair True True) = True
f _                = False

f' [x, y] | True <- x, True <- y = True
f' _                             = False
```

Note that the strictness of f differs from that of g defined below:

```
g [True, True] = True
g _            = False

*Main> f (False:undefined)
*** Exception: Prelude.undefined
*Main> g (False:undefined)
False
```

6.7.4.6 Pragmas for pattern synonyms

The *INLINABLE pragma* (page 612), *INLINE pragma* (page 609) and *NOINLINE pragma* (page 612) are supported for pattern synonyms as of GHC 9.2. For example:

```
pattern InlinablePattern x = [x]
{-# INLINABLE InlinablePattern #-}
pattern InlinedPattern x = [x]
{-# INLINE InlinedPattern #-}
pattern NonInlinedPattern x = [x]
{-# NOINLINE NonInlinedPattern #-}
```

As with other `INLINABLE`, `INLINE` and `NOINLINE` pragmas, it's possible to specify to which phase the pragma applies:

```
pattern Q x = [x]
{-# NOINLINE[1] Q #-}
```

The pragmas are applied both when the pattern is used as a matcher, and as a data constructor. For explicitly bidirectional pattern synonyms, the pragma must be at top level, not nested in the where clause. For example, this won't compile:

```
pattern HeadC x <- x:xs where
  HeadC x = [x]
  {-# INLINE HeadC #-}
```

but this will:

```
pattern HeadC x <- x:xs where
  HeadC x = [x]
{-# INLINE HeadC #-}
```

When no pragma is provided for a pattern, the inlining decision is made by GHC's own inlining heuristics.

6.8 Class and instances declarations

This section documents GHC's type-class extensions. There's lots of background in the paper Type classes: exploring the design space (Simon Peyton Jones, Mark Jones, Erik Meijer).

6.8.1 Multi-parameter type classes

MultiParamTypeClasses

Implies *ConstrainedClassMethods* (page 471)

Implied by *FunctionalDependencies* (page 475)

Since 6.8.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow the definition of typeclasses with more than one parameter.

Multi-parameter type classes are permitted, with extension *MultiParamTypeClasses* (page 470). For example:

```
class Collection c a where
    union :: c a -> c a -> c a
    ...etc.
```

6.8.2 Undecidable (or recursive) superclasses

UndecidableSuperClasses

Since 8.0.1

Allow all superclass constraints, including those that may result in non-termination of the typechecker.

The language extension *UndecidableSuperClasses* (page 470) allows much more flexible constraints in superclasses.

A class cannot generally have itself as a superclass. So this is illegal

```
class C a => D a where ...
class D a => C a where ...
```

GHC implements this test conservatively when type functions, or type variables, are involved. For example

```
type family F a :: Constraint
class F a => C a where ...
```

GHC will complain about this, because you might later add

```
type instance F Int = C Int
```

and now we'd be in a superclass loop. Here's an example involving a type variable

```
class f (C f) => C f
class c       => Id c
```

If we expanded the superclasses of `C Id` we'd get first `Id (C Id)` and thence `C Id` again.

But superclass constraints like these are sometimes useful, and the conservative check is annoying where no actual recursion is involved.

Moreover genuinely-recursive superclasses are sometimes useful. Here's a real-life example (#10318)

```haskell
class (Frac (Frac a) ~ Frac a,
       Fractional (Frac a),
       IntegralDomain (Frac a))
    => IntegralDomain a where
 type Frac a :: Type
```

Here the superclass cycle does terminate but it's not entirely straightforward to see that it does.

With the language extension *UndecidableSuperClasses* (page 470) GHC lifts all restrictions on superclass constraints. If there really *is* a loop, GHC will only expand it to finite depth.

6.8.3 Constrained class method types

ConstrainedClassMethods

Since 6.8.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Implied by *MultiParamTypeClasses* (page 470)

Allows the definition of further constraints on individual class methods.

Haskell 98 prohibits class method types to mention constraints on the class type variable, thus:

```haskell
class Seq s a where
  fromList :: [a] -> s a
  elem     :: Eq a => a -> s a -> Bool
```

The type of `elem` is illegal in Haskell 98, because it contains the constraint `Eq a`, which constrains only the class type variable (in this case `a`). More precisely, a constraint in a class method signature is rejected if

- The constraint mentions at least one type variable. So this is allowed:

```haskell
class C a where
  op1 :: HasCallStack => a -> a
  op2 :: (?x::Int) => Int -> a
```

- All of the type variables mentioned are bound by the class declaration, and none is locally quantified. Examples:

```haskell
class C a where
  op3 :: Eq a => a -> a    -- Rejected: constrains class variable only
  op4 :: D b => a -> b     -- Accepted: constrains a locally-quantified variable `b`
  op5 :: D (a,b) => a -> b -- Accepted: constrains a locally-quantified variable `b`
```

GHC lifts this restriction with language extension *ConstrainedClassMethods* (page 471). The restriction is a pretty stupid one in the first place, so *ConstrainedClassMethods* (page 471) is implied by *MultiParamTypeClasses* (page 470).

6.8.4 Default method signatures

DefaultSignatures

Since 7.2.1

Allows the definition of default method signatures in class definitions.

Haskell 98 allows you to define a default implementation when declaring a class:

```
class Enum a where
  enum :: [a]
  enum = []
```

The type of the `enum` method is `[a]`, and this is also the type of the default method. You can change the type of the default method by requiring a different context using the extension *DefaultSignatures* (page 472). For instance, if you have written a generic implementation of enumeration in a class `GEnum` with method `genum`, you can specify a default method that uses that generic implementation. But your default implementation can only be used if the constraints are satisfied, therefore you need to change the type of the default method

```
class Enum a where
  enum :: [a]
  default enum :: (Generic a, GEnum (Rep a)) => [a]
  enum = map to genum
```

We reuse the keyword `default` to signal that a signature applies to the default method only; when defining instances of the `Enum` class, the original type `[a]` of `enum` still applies. When giving an empty instance, however, the default implementation `(map to genum)` is filled-in, and type-checked with the type `(Generic a, GEnum (Rep a)) => [a]`.

The type signature for a default method of a type class must take on the same form as the corresponding main method's type signature. Otherwise, the typechecker will reject that class's definition. By "take on the same form", we mean that the default type signature should differ from the main type signature only in their outermost contexts. Therefore, if you have a method `bar`:

```
class Foo a where
  bar :: forall b. C => a -> b -> b
```

Then a default method for `bar` must take on the form:

```
default bar :: forall b. C' => a -> b -> b
bar = ...
```

`C` is allowed to be different from `C'`, but the right-hand sides of the type signatures must coincide. We require this because when you declare an empty instance for a class that uses *DefaultSignatures* (page 472), GHC implicitly fills in the default implementation like this:

```
instance Foo Int where
  bar = default_bar
```

Where `default_bar` is a top-level function based on the default type signature and implementation for `bar`:

```
default_bar :: forall a b. (Foo a, C') => a -> b -> b
default_bar = ...
```

In order for this approach to work, the default type signature for `bar` should be the same as the non-default signature, modulo the outermost context (with some caveats—see *Detailed requirements for default type signatures* (page 473)). There is no obligation for `C` and `C'` to be the same, and indeed, the `Enum` example above relies on `enum`'s default type signature having a more specific context than the original type signature.

We use default signatures to simplify generic programming in GHC (*Generic programming* (page 597)).

6.8.5 Detailed requirements for default type signatures

The rest of this section gives further details about what constitutes valid default type signatures.

- Ignoring outermost contexts, a default type signature must match the original type signature according to *GHC's subsumption rules* (page 404). As a result, the order of type variables in the default signature is important. Recall the `Foo` example from the previous section:

  ```
  class Foo a where
    bar :: forall b. C => a -> b -> b

    default bar :: forall b. C' => a -> b -> b
    bar = ...
  ```

  This is legal because if you remove the outermost contexts `C` and `C'`, then the two type signatures are the same. It is not necessarily the case that the default signature has to be *exactly* the same, however. For instance, this would also be an acceptable default type signature, as it is alpha-equivalent to the original type signature:

  ```
  default bar :: forall x. C' => a -> x -> x
  ```

  On the other hand, this is *not* an acceptable default type signature, since the type variable `a` is in the wrong place:

  ```
  default bar :: forall b. C' => b -> a -> b
  ```

- The only place where a default type signature is allowed to more precise than the original type signature is in the outermost context. For example, this would *not* be an acceptable default type signature, since we can't match the type variable `b` with the concrete type `Int`:

  ```
  default bar :: C' => a -> Int -> Int
  ```

  You can, however, use type equalities to achieve the same result:

  ```
  default bar :: forall b. (C', b ~ Int) => a -> b -> b
  ```

- Because of *GHC's subsumption rules* (page 404) rules, there are relatively tight restrictions concerning nested or higher-rank `forall`s (see *Arbitrary-rank polymorphism* (page 402)). Consider this class:

  ```
  class C x where
    m :: x -> forall a b. a -> b
  ```

  GHC would *not* permit the following default type signature for `m`:

```
default m :: x -> forall b a. a -> b
```

This is because the default signature quantifies the nested `forall`s in a different order than the original type signature. In order for this to typecheck, the default signature must preserve the original order:

```
default m :: x -> forall a b. a -> b
```

Note that unlike nested or higher-rank `forall`s, outermost `forall`s have more flexibility in how they are ordered. As a result, GHC will permit the following:

```
class C' x where
  m'         :: forall a b. x -> a -> b
  default m' :: forall b a. x -> a -> b
  m' = ...
```

- Just as the order of nested or higher-rank `forall`s is restricted, a similar restriction applies to the order in which nested or higher-rank contexts appear. As a result, GHC will not permit the following:

```
class D a where
  n         :: a -> forall b. (Eq b, Show b) => b -> String
  default n :: a -> forall b. (Show b, Eq b) => b -> String
  n = ...
```

GHC will permit reordering constraints within an outermost context, however, as demonstrated by the fact that GHC accepts the following:

```
class D' a where
  n'         :: (Eq b, Show b) => a -> b -> String
  default n' :: (Show b, Eq b) => a -> b -> String
  n' = ...
```

- Because a default signature is only ever allowed to differ from its original type signature in the outermost context, not in nested or higher-rank contexts, there are certain defaults that cannot be written without reordering `forall` s. Consider this example:

```
class E a where
  p :: Int -> forall b. b -> String
```

Suppose one wishes to write a default signature for `p` where the context must mention both `a` and `b`. While the natural thing to do would be to write this default:

```
default p :: Int -> forall b. DefaultClass a b => b -> String
```

This will not typecheck, since the default type signature now differs from the original type signature in its use of nested contexts. The only way to make such a default signature work is to change the order in which `b` is quantified:

```
default p :: forall b. DefaultClass a b => Int -> b -> String
```

This works, but at the expense of changing `p`'s behavior with respect to *Visible type application* (page 386).

6.8.6 Nullary type classes

NullaryTypeClasses

Since 7.8.1

Status Deprecated

Allow use and definition of type classes with no parameters. This extension has been replaced by *MultiParamTypeClasses* (page 470).

Nullary (no parameter) type classes are enabled with *MultiParamTypeClasses* (page 470); historically, they were enabled with the (now deprecated) *NullaryTypeClasses* (page 475). Since there are no available parameters, there can be at most one instance of a nullary class. A nullary type class might be used to document some assumption in a type signature (such as reliance on the Riemann hypothesis) or add some globally configurable settings in a program. For example,

```
class RiemannHypothesis where
  assumeRH :: a -> a

-- Deterministic version of the Miller test
-- correctness depends on the generalised Riemann hypothesis
isPrime :: RiemannHypothesis => Integer -> Bool
isPrime n = assumeRH (...)
```

The type signature of `isPrime` informs users that its correctness depends on an unproven conjecture. If the function is used, the user has to acknowledge the dependence with:

```
instance RiemannHypothesis where
  assumeRH = id
```

6.8.7 Functional dependencies

FunctionalDependencies

Implies *MultiParamTypeClasses* (page 470)

Since 6.8.1

Allow use of functional dependencies in class declarations.

Functional dependencies are implemented as described by Mark Jones in [Jones2000].

Functional dependencies are introduced by a vertical bar in the syntax of a class declaration; e.g.

```
class (Monad m) => MonadState s m | m -> s where ...

class Foo a b c | a b -> c where ...
```

More documentation can be found in the Haskell Wiki.

6.8.7.1 Rules for functional dependencies

In a class declaration, all of the class type variables must be reachable (in the sense mentioned in *Loosening restrictions on class contexts* (page 499)) from the free variables of each method type. For example:

```
class Coll s a where
  empty  :: s
  insert :: s -> a -> s
```

is not OK, because the type of `empty` doesn't mention `a`. Functional dependencies can make the type variable reachable:

```
class Coll s a | s -> a where
  empty  :: s
  insert :: s -> a -> s
```

Alternatively `Coll` might be rewritten

```
class Coll s a where
  empty  :: s a
  insert :: s a -> a -> s a
```

which makes the connection between the type of a collection of `a`'s (namely `(s a)`) and the element type `a`. Occasionally this really doesn't work, in which case you can split the class like this:

```
class CollE s where
  empty  :: s

class CollE s => Coll s a where
  insert :: s -> a -> s
```

6.8.7.2 Background on functional dependencies

The following description of the motivation and use of functional dependencies is taken from the Hugs user manual, reproduced here (with minor changes) by kind permission of Mark Jones.

Consider the following class, intended as part of a library for collection types:

```
class Collects e ce where
    empty  :: ce
    insert :: e -> ce -> ce
    member :: e -> ce -> Bool
```

The type variable `e` used here represents the element type, while `ce` is the type of the container itself. Within this framework, we might want to define instances of this class for lists or characteristic functions (both of which can be used to represent collections of any equality type), bit sets (which can be used to represent collections of characters), or hash tables (which can be used to represent any collection whose elements have a hash function). Omitting standard implementation details, this would lead to the following declarations:

```
instance Eq e => Collects e [e] where ...
instance Eq e => Collects e (e -> Bool) where ...
```

(continues on next page)

(continued from previous page)

```
instance Collects Char BitSet where ...
instance (Hashable e, Collects a ce)
           => Collects e (Array Int ce) where ...
```

All this looks quite promising; we have a class and a range of interesting implementations. Unfortunately, there are some serious problems with the class declaration. First, the empty function has an ambiguous type:

```
empty :: Collects e ce => ce
```

By "ambiguous" we mean that there is a type variable `e` that appears on the left of the `=>` symbol, but not on the right. The problem with this is that, according to the theoretical foundations of Haskell overloading, we cannot guarantee a well-defined semantics for any term with an ambiguous type.

We can sidestep this specific problem by removing the empty member from the class declaration. However, although the remaining members, insert and member, do not have ambiguous types, we still run into problems when we try to use them. For example, consider the following two functions:

```
f x y = insert x . insert y
g     = f True 'a'
```

for which GHC infers the following types:

```
f :: (Collects a c, Collects b c) => a -> b -> c -> c
g :: (Collects Bool c, Collects Char c) => c -> c
```

Notice that the type for `f` allows the two parameters `x` and `y` to be assigned different types, even though it attempts to insert each of the two values, one after the other, into the same collection. If we're trying to model collections that contain only one type of value, then this is clearly an inaccurate type. Worse still, the definition for `g` is accepted, without causing a type error. As a result, the error in this code will not be flagged at the point where it appears. Instead, it will show up only when we try to use `g`, which might even be in a different module.

An attempt to use constructor classes

Faced with the problems described above, some Haskell programmers might be tempted to use something like the following version of the class declaration:

```
class Collects e c where
   empty  :: c e
   insert :: e -> c e -> c e
   member :: e -> c e -> Bool
```

The key difference here is that we abstract over the type constructor `c` that is used to form the collection type `c e`, and not over that collection type itself, represented by `ce` in the original class declaration. This avoids the immediate problems that we mentioned above: empty has type `Collects e c => c e`, which is not ambiguous.

The function `f` from the previous section has a more accurate type:

```
f :: (Collects e c) => e -> e -> c e -> c e
```

The function g from the previous section is now rejected with a type error as we would hope because the type of f does not allow the two arguments to have different types. This, then, is an example of a multiple parameter class that does actually work quite well in practice, without ambiguity problems. There is, however, a catch. This version of the Collects class is nowhere near as general as the original class seemed to be: only one of the four instances for Collects given above can be used with this version of Collects because only one of them—the instance for lists—has a collection type that can be written in the form c e, for some type constructor c, and element type e.

Adding functional dependencies

To get a more useful version of the Collects class, GHC provides a mechanism that allows programmers to specify dependencies between the parameters of a multiple parameter class (For readers with an interest in theoretical foundations and previous work: The use of dependency information can be seen both as a generalisation of the proposal for "parametric type classes" that was put forward by Chen, Hudak, and Odersky, or as a special case of Mark Jones's later framework for "improvement" of qualified types. The underlying ideas are also discussed in a more theoretical and abstract setting in a manuscript [Jones1999], where they are identified as one point in a general design space for systems of implicit parameterisation). To start with an abstract example, consider a declaration such as:

```
class C a b where ...
```

which tells us simply that C can be thought of as a binary relation on types (or type constructors, depending on the kinds of a and b). Extra clauses can be included in the definition of classes to add information about dependencies between parameters, as in the following examples:

```
class D a b | a -> b where ...
class E a b | a -> b, b -> a where ...
```

The notation a -> b used here between the | and where symbols — not to be confused with a function type — indicates that the a parameter uniquely determines the b parameter, and might be read as "a determines b." Thus D is not just a relation, but actually a (partial) function. Similarly, from the two dependencies that are included in the definition of E, we can see that E represents a (partial) one-to-one mapping between types.

More generally, dependencies take the form x1 ... xn -> y1 ... ym, where x1, ..., xn, and y1, ..., yn are type variables with n>0 and m>=0, meaning that the y parameters are uniquely determined by the x parameters. Spaces can be used as separators if more than one variable appears on any single side of a dependency, as in t -> a b. Note that a class may be annotated with multiple dependencies using commas as separators, as in the definition of E above. Some dependencies that we can write in this notation are redundant, and will be rejected because they don't serve any useful purpose, and may instead indicate an error in the program. Examples of dependencies like this include a -> a, a -> a a, a ->, etc. There can also be some redundancy if multiple dependencies are given, as in a->b, b->c, a->c, and in which some subset implies the remaining dependencies. Examples like this are not treated as errors. Note that dependencies appear only in class declarations, and not in any other part of the language. In particular, the syntax for instance declarations, class constraints, and types is completely unchanged.

By including dependencies in a class declaration, we provide a mechanism for the programmer to specify each multiple parameter class more precisely. The compiler, on the other hand, is responsible for ensuring that the set of instances that are in scope at any given point in the program is consistent with any declared dependencies. For example, the following pair of instance declarations cannot appear together in the same scope because they violate the dependency for D, even though either one on its own would be acceptable:

```
instance D Bool Int where ...
instance D Bool Char where ...
```

Note also that the following declaration is not allowed, even by itself:

```
instance D [a] b where ...
```

The problem here is that this instance would allow one particular choice of [a] to be associated with more than one choice for b, which contradicts the dependency specified in the definition of D. More generally, this means that, in any instance of the form:

```
instance D t s where ...
```

for some particular types t and s, the only variables that can appear in s are the ones that appear in t, and hence, if the type t is known, then s will be uniquely determined.

The benefit of including dependency information is that it allows us to define more general multiple parameter classes, without ambiguity problems, and with the benefit of more accurate types. To illustrate this, we return to the collection class example, and annotate the original definition of Collects with a simple dependency:

```
class Collects e ce | ce -> e where
   empty  :: ce
   insert :: e -> ce -> ce
   member :: e -> ce -> Bool
```

The dependency ce -> e here specifies that the type e of elements is uniquely determined by the type of the collection ce. Note that both parameters of Collects are of kind Type; there are no constructor classes here. Note too that all of the instances of Collects that we gave earlier can be used together with this new definition.

What about the ambiguity problems that we encountered with the original definition? The empty function still has type Collects e ce => ce, but it is no longer necessary to regard that as an ambiguous type: Although the variable e does not appear on the right of the => symbol, the dependency for class Collects tells us that it is uniquely determined by ce, which does appear on the right of the => symbol. Hence the context in which empty is used can still give enough information to determine types for both ce and e, without ambiguity. More generally, we need only regard a type as ambiguous if it contains a variable on the left of the => that is not uniquely determined (either directly or indirectly) by the variables on the right.

Dependencies also help to produce more accurate types for user defined functions, and hence to provide earlier detection of errors, and less cluttered types for programmers to work with. Recall the previous definition for a function f:

```
f x y = insert x y = insert x . insert y
```

for which we originally obtained a type:

```
f :: (Collects a c, Collects b c) => a -> b -> c -> c
```

Given the dependency information that we have for Collects, however, we can deduce that a and b must be equal because they both appear as the second parameter in a Collects constraint with the same first parameter c. Hence we can infer a shorter and more accurate type for f:

```
f :: (Collects a c) => a -> a -> c -> c
```

In a similar way, the earlier definition of `g` will now be flagged as a type error.

Although we have given only a few examples here, it should be clear that the addition of dependency information can help to make multiple parameter classes more useful in practice, avoiding ambiguity problems, and allowing more general sets of instance declarations.

6.8.8 Instance declarations and resolution

An instance declaration has the form

```
instance (assertion1, ..., assertionn) => class type1 ... typem where ...
```

The part before the "=>" is the *context*, while the part after the "=>" is the *head* of the instance declaration.

When GHC tries to resolve, say, the constraint `C Int Bool`, it tries to match every instance declaration against the constraint, by instantiating the head of the instance declaration. Consider these declarations:

```
instance context1 => C Int a     where ...  -- (A)
instance context2 => C a   Bool  where ...  -- (B)
```

GHC's default behaviour is that *exactly one instance must match the constraint it is trying to resolve*. For example, the constraint `C Int Bool` matches instances (A) and (B), and hence would be rejected; while `C Int Char` matches only (A) and hence (A) is chosen.

Notice that

- When matching, GHC takes no account of the context of the instance declaration (`context1` etc).
- It is fine for there to be a *potential* of overlap (by including both declarations (A) and (B), say); an error is only reported if a particular constraint matches more than one.

See also *Overlapping instances* (page 486) for flags that loosen the instance resolution rules.

6.8.8.1 Relaxed rules for the instance head

`TypeSynonymInstances`

> **Implied by** *`FlexibleInstances`* (page 480)
>
> **Since** 6.8.1
>
> **Status** Included in *`GHC2024`* (page 270), *`GHC2021`* (page 271)

Allow definition of type class instances for type synonyms.

`FlexibleInstances`

> **Implies** *`TypeSynonymInstances`* (page 480)
>
> **Since** 6.8.1
>
> **Status** Included in *`GHC2024`* (page 270), *`GHC2021`* (page 271)

Allow definition of type class instances with arbitrary nested types in the instance head.

In Haskell 98 the head of an instance declaration must be of the form `C (T a1 ... an)`, where `C` is the class, `T` is a data type constructor, and the `a1 ... an` are distinct type variables. In the case of multi-parameter type classes, this rule applies to each parameter of the instance head (Arguably it should be okay if just one has this form and the others are type variables, but that's the rules at the moment).

GHC relaxes this rule in two ways:

- With the *TypeSynonymInstances* (page 480) extension, instance heads may use type synonyms. As always, using a type synonym is just shorthand for writing the RHS of the type synonym definition. For example:

  ```
  type Point a = (a,a)
  instance C (Point a) where ...
  ```

  is legal. The instance declaration is equivalent to

  ```
  instance C (a,a) where ...
  ```

  As always, type synonyms must be fully applied. You cannot, for example, write:

  ```
  instance Monad Point where ...
  ```

- The *FlexibleInstances* (page 480) extension allows the head of the instance declaration to mention arbitrary nested types. For example, this becomes a legal instance declaration

  ```
  instance C (Maybe Int) where ...
  ```

  See also the *rules on overlap* (page 486).

  The *FlexibleInstances* (page 480) extension implies *TypeSynonymInstances* (page 480).

However, the instance declaration must still conform to the rules for instance termination: see *Instance termination rules* (page 483).

6.8.8.2 Formal syntax for instance declaration types

The top of an instance declaration only permits very specific forms of types. To make more precise what forms of types are or are not permitted, we provide a BNF-style grammar for the tops of instance declarations below.

```
inst_top ::= 'instance' opt_forall opt_ctxt inst_head opt_where

opt_forall ::= <empty>
            |  'forall' tv_bndrs '.'

tv_bndrs ::= <empty>
          |  tv_bndr tv_bndrs

tv_bndr ::= tyvar
         |  '(' tyvar '::' ctype ')'

opt_ctxt ::= <empty>
          |  btype '=>'
          |  '(' ctxt ')' '=>'
```

(continues on next page)

(continued from previous page)

```
ctxt ::= ctype
      |  ctype ',' ctxt

inst_head ::= '(' inst_head ')'
           |  prefix_cls_tycon arg_types
           |  arg_type infix_cls_tycon arg_type
           |  '(' arg_type infix_cls_tycon arg_type ')' arg_types

arg_types ::= <empty>
           |  arg_type arg_types

opt_where ::= <empty>
           |  'where'
```

Where:

- `btype` is a type that is not allowed to have an outermost `forall`/`=>` unless it is surrounded by parentheses. For example, `forall a. a` and `Eq a => a` are not legal `btype`s, but `(forall a. a)` and `(Eq a => a)` are legal.
- `ctype` is a `btype` that has no restrictions on an outermost `forall`/`=>`, so `forall a. a` and `Eq a => a` are legal `ctype`s.
- `arg_type` is a type that is not allowed to have `forall`s or `=>`s
- `prefix_cls_tycon` is a class type constructor written prefix (e.g., `Show` or `(&&&)`), while `infix_cls_tycon` is a class type constructor written infix (e.g., \`Show\` or `&&&`).

This is a simplified grammar that does not fully delve into all of the implementation details of GHC's parser (such as the placement of Haddock comments), but it is sufficient to attain an understanding of what is syntactically allowed. Some further various observations about this grammar:

- Instance declarations are not allowed to be declared with nested `forall`s or `=>`s. For example, this would be rejected:

  ```
  instance forall a. forall b. C (Either a b) where ...
  ```

  As a result, `inst_top` puts all of its quantification and constraints up front with `opt_forall` and `opt_context`.
- Furthermore, instance declarations types do not permit outermost parentheses that surround the `opt_forall` or `opt_ctxt`, if at least one of them are used. For example, `instance (forall a. C a)` would be rejected, since GHC would treat the `forall` as being nested.

  Note that it is acceptable to use parentheses in a `inst_head`. For instance, `instance (C a)` is accepted, as is `instance forall a. (C a)`.

6.8.8.3 Instance termination rules

Regardless of *FlexibleInstances* (page 480) and *FlexibleContexts* (page 499), instance declarations must conform to some rules that ensure that instance resolution will terminate. The restrictions can be lifted with *UndecidableInstances* (page 484) (see *Undecidable instances and loopy superclasses* (page 484)).

The rules are these:

1. The Paterson Conditions: for each class constraint `(C t1 ... tn)` in the context
   1. No type variable has more occurrences in the constraint than in the head
   2. The constraint has fewer constructors and variables (taken together and counting repetitions) than the head
   3. The constraint mentions no type functions. A type function application can in principle expand to a type of arbitrary size, and so are rejected out of hand

   If these three conditions hold we say that the constraint `(C t1 ... tn)` is **Paterson-smaller** than the instance head.
2. The Coverage Condition. For each functional dependency, $\langle tvs \rangle_{left}$ `->` $\langle tvs \rangle_{right}$, of the class, every type variable in $S(\langle tvs \rangle_{right})$ must appear in $S(\langle tvs \rangle_{left})$, where S is the substitution mapping each type variable in the class declaration to the corresponding type in the instance head.

These restrictions ensure that instance resolution terminates: each reduction step makes the problem smaller by at least one constructor. You can find lots of background material about the reason for these restrictions in the paper Understanding functional dependencies via Constraint Handling Rules.

For example, these are okay:

```
instance C Int [a]          -- Multiple parameters
instance Eq (S [a])         -- Structured type in head

    -- Repeated type variable in head
instance C4 a a => C4 [a] [a]
instance Stateful (ST s) (MutVar s)

    -- Head can consist of type variables only
instance C a
instance (Eq a, Show b) => C2 a b

    -- Non-type variables in context
instance Show (s a) => Show (Sized s a)
instance C2 Int a => C3 Bool [a]
instance C2 Int a => C3 [a] b
```

But these are not:

```
    -- Context assertion no smaller than head
instance C a => C a where ...
    -- (C b b) has more occurrences of b than the head
instance C b b => Foo [b] where ...
```

The same restrictions apply to instances generated by `deriving` clauses. Thus the following is accepted:

```
data MinHeap h a = H a (h a)
  deriving (Show)
```

because the derived instance

```
instance (Show a, Show (h a)) => Show (MinHeap h a)
```

conforms to the above rules.

The restrictions on functional dependencies (*Functional dependencies* (page 475)) are particularly troublesome. It is tempting to introduce type variables in the context that do not appear in the head, something that is excluded by the normal rules. For example:

```
class HasConverter a b | a -> b where
   convert :: a -> b

data Foo a = MkFoo a

instance (HasConverter a b,Show b) => Show (Foo a) where
   show (MkFoo value) = show (convert value)
```

This is dangerous territory, however. Here, for example, is a program that would make the typechecker loop:

```
class D a
class F a b | a->b
instance F [a] [[a]]
instance (D c, F a c) => D [a]   -- 'c' is not mentioned in the head
```

Similarly, it can be tempting to lift the coverage condition:

```
class Mul a b c | a b -> c where
  (.*.) :: a -> b -> c

instance Mul Int Int Int where (.*.) = (*)
instance Mul Int Float Float where x .*. y = fromIntegral x * y
instance Mul a b c => Mul a [b] [c] where x .*. v = map (x.*.) v
```

The third instance declaration does not obey the coverage condition; and indeed the (somewhat strange) definition:

```
f = \ b x y -> if b then x .*. [y] else y
```

makes instance inference go into a loop, because it requires the constraint `(Mul a [b] b)`.

6.8.8.4 Undecidable instances and loopy superclasses

UndecidableInstances

> **Since** 6.8.1
>
> Permit definition of instances which may lead to type-checker non-termination.

The `UndecidableInstances` (page 484) extension lifts the restrictions on on instance declarations described in *Instance termination rules* (page 483). The `UndecidableInstances` (page 484) extension also lifts some of the restrictions imposed on type family instances; see *Decidability of type synonym instances* (page 346).

With *UndecidableInstances* (page 484) it is possible to create a superclass cycle, which leads to the program failing to terminate. To avoid this, GHC imposes rules on the way in which superclass constraints are satisfied in an instance declaration. These rules apply even when *UndecidableInstances* (page 484) is enabled. Consider:

```
class C a => D a where ...

instance Wombat [b] => D [b] where ...
```

When typechecking this `instance` declaration, GHC must ensure that `D`'s superclass, `(C [b])` is satisfied. We say that `(C [b])` is a **Wanted superclass constraint** of the instance declaration.

If there is an `instance blah => C [b]`, which is often the case, GHC can use the instance declaration and all is well. But suppose there is no such instance, so GHC can only satisfy the Wanted `(C [b])` from the context of the instance, namely the Given constraint `(Wombat [b])`. Perhaps the declaration of `Wombat` looks like this:

```
class C a => Wombat a
```

So the Given `(Wombat [b])` has a superclass `(C [b])`, and it looks as if we can satisfy the Wanted `(C [b])` constraint from this superclass of `Wombat`. But it turns out that allowing this can lead to subtle looping dictionaries, and GHC prevents it.

The rule is this: **a Wanted superclass constraint can only be satisfied in one of these three ways:**

1. *Directly from the context of the instance declaration*. For example, if the declaration looked like this:

   ```
   instance (Wombat [b], C [b]) => D [b] where ...
   ```

   we could satisfy the Wanted `(C [b])` from the Given `(C [b])`.

2. *Using another instance declaration*. For example, if we had:

   ```
   instance C b => C [b] where ...
   ```

   we can satisfy the Wanted superclass constraint `(C [b])` using this instance, reducing it to the Wanted constraint `(C b)` (which still has to be solved).

3. *Using the immediate superclass of a Given constraint X that is Paterson-smaller than the head of the instance declaration.* The rules for Paterson-smaller are precisely those described in *Instance termination rules* (page 483):

   - No type variable can occur more often in X than in the instance head.
   - X must have fewer type constructors and variables (taken together and counting repetitions) than the instance head.
   - X must mention no type functions.

Rule (3) is the tricky one. Here is an example, taken from GHC's own source code:

```
       class Ord r => UserOfRegs r a where ...
(i1)   instance UserOfRegs r a => UserOfRegs r (Maybe a) where
(i2)   instance (Ord r, UserOfRegs r CmmReg) => UserOfRegs r CmmExpr where
```

For `(i1)` we can get the `(Ord r)` superclass by selection from `(UserOfRegs r a)`, since it (i.e. `UserOfRegs r a`) is Paterson-smaller than the head of the instance declaration, namely `(UserOfRegs r (Maybe a))`.

But for `(i2)` that isn't the case: `(UserOfRegs r CmmReg)` is not Paterson-smaller than the head of the instance `(UserOfRegs r CmmExpr)`, so we can't use the superclasses of the former. Hence we must instead add an explicit, and perhaps surprising, `(Ord r)` argument to the instance declaration.

This fix, of simply adding an apparently-redundant constraint to the context of an instance declaration, is robust: it always fixes the problem. (We considered adding it automatically, but decided that it was better be explicit.)

Fixing this subtle superclass cycle has a long history; if you are interested, read `Note [Recursive superclasses]` and `Note [Solving superclass constraints]` in `GHC.Tc.TyCl.Instance`.

6.8.8.5 Overlapping instances

`OverlappingInstances`

> **Since** 6.8.1
>
> **Status** Deprecated

Deprecated extension to weaken checks intended to ensure instance resolution termination.

`IncoherentInstances`

> **Since** 6.8.1
>
> **Status** Deprecated

Deprecated extension to weaken checks intended to ensure instance resolution termination.

In general, as discussed in *Instance declarations and resolution* (page 480), *GHC requires that it be unambiguous which instance declaration should be used to resolve a type-class constraint*. GHC also provides a way to loosen the instance resolution, by allowing more than one instance to match, *provided there is a most specific one*. Moreover, it can be loosened further, by allowing more than one instance to match irrespective of whether there is a most specific one. This section gives the details.

To control the choice of instance, it is possible to specify the overlap behavior for individual instances with a pragma, written immediately after the `instance` keyword. The pragma may be one of: `{-# OVERLAPPING #-}`, `{-# OVERLAPPABLE #-}`, `{-# OVERLAPS #-}`, or `{-# INCOHERENT #-}`.

The matching behaviour is also influenced by two module-level language extension flags: *`OverlappingInstances`* (page 486) and *`IncoherentInstances`* (page 486). These extensions are now deprecated (since GHC 7.10) in favour of the fine-grained per-instance pragmas.

A more precise specification is as follows. The willingness to be overlapped or incoherent is a property of the *instance declaration* itself, controlled as follows:

- An instance is *incoherent* if: it has an `INCOHERENT` pragma; or if the instance has no pragma and it appears in a module compiled with *`IncoherentInstances`* (page 486).

- An instance is *overlappable* if: it has an `OVERLAPPABLE` or `OVERLAPS` pragma; or if the instance has no pragma and it appears in a module compiled with *OverlappingInstances* (page 486); or if the instance is incoherent.
- An instance is *overlapping* if: it has an `OVERLAPPING` or `OVERLAPS` pragma; or if the instance has no pragma and it appears in a module compiled with *OverlappingInstances* (page 486); or if the instance is incoherent.

Now suppose that, in some client module, we are searching for an instance of the *target constraint* `(C ty1 .. tyn)`. The search works like this:

- Find all instances I that *match* the target constraint; that is, the target constraint is a substitution instance of I. These instance declarations are the *candidates*.
- If no candidates remain, the search fails
- Eliminate any candidate IX for which there is another candidate IY such that both of the following hold:
  - IY is strictly more specific than IX. That is, IY is a substitution instance of IX but not vice versa.
  - IX is *overlappable* or IY is *overlapping*. (This "or" design, rather than an "and" design, allows a client to deliberately override an instance from a library, without requiring a change to the library.)
- If all the remaining candidates are incoherent, the search succeeds, returning an arbitrary surviving candidate.
- If more than one non-incoherent candidate remains, the search fails.
- Otherwise there is exactly one non-incoherent candidate; call it the "prime candidate".
- Now find all instances, or in-scope given constraints, that *unify* with the target constraint, but do not *match* it. Such non-candidate instances might match when the target constraint is further instantiated. If all of them are incoherent top-level instances, the search succeeds, returning the prime candidate. Otherwise the search fails.

Notice that these rules are not influenced by flag settings in the client module, where the instances are *used*. These rules make it possible for a library author to design a library that relies on overlapping instances without the client having to know.

Errors are reported *lazily* (when attempting to solve a constraint), rather than *eagerly* (when the instances themselves are defined). Consider, for example

```
instance C Int  b where ..
instance C a Bool where ..
```

These potentially overlap, but GHC will not complain about the instance declarations themselves, regardless of flag settings. If we later try to solve the constraint `(C Int Char)` then only the first instance matches, and all is well. Similarly with `(C Bool Bool)`. But if we try to solve `(C Int Bool)`, both instances match and an error is reported.

As a more substantial example of the rules in action, consider

```
instance {-# OVERLAPPABLE #-} context1 => C Int b     where ...  -- (A)
instance {-# OVERLAPPABLE #-} context2 => C a   Bool  where ...  -- (B)
instance {-# OVERLAPPABLE #-} context3 => C a   [b]   where ...  -- (C)
instance {-# OVERLAPPING  #-} context4 => C Int [Int] where ...  -- (D)
```

(These all need *FlexibleInstances* (page 480).) Now suppose that the type inference engine needs to solve the constraint `C Int [Int]`. This constraint matches instances (A), (C) and (D), but the last is more specific, and hence is chosen.

If (D) did not exist then (A) and (C) would still be matched, but neither is most specific. In that case, the program would be rejected, unless *IncoherentInstances* (page 486) is enabled, in which case it would be accepted and (A) or (C) would be chosen arbitrarily.

An instance declaration is *more specific* than another iff the head of former is a substitution instance of the latter. For example (D) is "more specific" than (C) because you can get from (C) to (D) by substituting `a := Int` and `b := Int`.

The final bullet (about unifying instances) makes GHC conservative about committing to an overlapping instance. For example:

```
f :: [b] -> [b]
f x = ...
```

Suppose that from the RHS of `f` we get the constraint `C b [b]`. But GHC does not commit to instance (C), because in a particular call of `f`, `b` might be instantiated to `Int`, in which case instance (D) would be more specific still. So GHC rejects the program.

If, however, you enable the extension *IncoherentInstances* (page 486) when compiling the module that contains (D), GHC will instead pick (C), without complaining about the problem of subsequent instantiations.

Notice that we gave a type signature to `f`, so GHC had to *check* that `f` has the specified type. Suppose instead we do not give a type signature, asking GHC to *infer* it instead. In this case, GHC will refrain from simplifying the constraint `C Int [b]` (for the same reason as before) but, rather than rejecting the program, it will infer the type

```
f :: C b [b] => [b] -> [b]
```

That postpones the question of which instance to pick to the call site for `f` by which time more is known about the type `b`. You will need the *FlexibleContexts* (page 499) extension.

Exactly the same situation can arise in instance declarations themselves. Suppose we have

```
class Foo a where
   f :: a -> a
instance Foo [b] where
   f x = ...
```

and, as before, the constraint `C Int [b]` arises from `f`'s right hand side. GHC will reject the instance, complaining as before that it does not know how to resolve the constraint `C Int [b]`, because it matches more than one instance declaration. The solution is to postpone the choice by adding the constraint to the context of the instance declaration, thus:

```
instance C Int [b] => Foo [b] where
   f x = ...
```

(You need *FlexibleContexts* (page 499) to do this.)

In the unification check in the final bullet, GHC also uses the "in-scope given constraints". Consider for example

```
instance C a Int
```

(continues on next page)

(continued from previous page)

```
g :: forall b c. C b Int => blah
g = ...needs (C c Int)...
```

Here GHC will not solve the constraint `(C c Int)` from the top-level instance, because a particular call of `g` might instantiate both `b` and `c` to the same type, which would allow the constraint to be solved in a different way. This latter restriction is principally to make the constraint-solver complete. (Interested folk can read `Note [Instance and Given overlap]` in `TcInteract`.) It is easy to avoid: in a type signature avoid a constraint that matches a top-level instance. The flag *-Wsimplifiable-class-constraints* (page 101) warns about such signatures.

Warning: Overlapping instances must be used with care. They can give rise to incoherence (i.e. different instance choices are made in different parts of the program) even without *IncoherentInstances* (page 486). Consider:

```
{-# LANGUAGE OverlappingInstances #-}
module Help where

    class MyShow a where
    myshow :: a -> String

    instance MyShow a => MyShow [a] where
    myshow xs = concatMap myshow xs

    showHelp :: MyShow a => [a] -> String
    showHelp xs = myshow xs

{-# LANGUAGE FlexibleInstances, OverlappingInstances #-}
module Main where
    import Help

    data T = MkT

    instance MyShow T where
    myshow x = "Used generic instance"

    instance MyShow [T] where
    myshow xs = "Used more specific instance"

    main = do { print (myshow [MkT]); print (showHelp [MkT]) }
```

In function `showHelp` GHC sees no overlapping instances, and so uses the `MyShow [a]` instance without complaint. In the call to `myshow` in `main`, GHC resolves the `MyShow [T]` constraint using the overlapping instance declaration in module `Main`. As a result, the program prints

```
"Used more specific instance"
"Used generic instance"
```

(An alternative possible behaviour, not currently implemented, would be to reject module `Help` on the grounds that a later instance declaration might overlap the local one.)

6.8.8.6 Instance signatures: type signatures in instance declarations

InstanceSigs

> **Since** 7.6.1
>
> **Status** Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow type signatures for members in instance definitions.

The *InstanceSigs* (page 490) extension allows users to give type signatures to the class methods in a class instance declaration. For example:

```
data T a = MkT a a
instance Eq a => Eq (T a) where
  (==) :: T a -> T a -> Bool   -- The instance signature
  (==) (MkT x1 x2) (MkTy y1 y2) = x1==y1 && x2==y2
```

Some details:

- The type signature in the instance declaration must be more polymorphic than (or the same as) the one in the class declaration, instantiated with the instance type. For example, this is fine:

  ```
  instance Eq a => Eq (T a) where
     (==) :: forall b. b -> b -> Bool
     (==) x y = True
  ```

  Here the signature in the instance declaration is more polymorphic than that required by the instantiated class method.

  Note that, to check that the instance signature is more polymorphic, GHC performs a sub-type check, which can solve constraints using available top-level instances. This means that the following instance signature is accepted:

  ```
  instance Eq (T Int) where
    (==) :: Eq Int => T Int -> T Int -> Bool
    (==) (MkT x1 _) (MkT y1 _) = x1 == y1
  ```

  The `Eq Int` constraint in the instance signature will be solved by the top-level `Eq Int` instance, from which it follows that the instance signature is indeed as general as the instantiated class method type `T Int -> T Int -> Bool`.

- The code for the method in the instance declaration is typechecked against the type signature supplied in the instance declaration, as you would expect. So if the instance signature is more polymorphic than required, the code must be too.

- The instance signature is purely local to the class instance declaration. It only affects the typechecking of the method in the instance; it does not affect anything outside the class instance. In this way, it is similar to an inline type signature:

  ```
  instance Eq a => Eq (T a) where
     (==) = (\ x y -> True) :: forall b. b -> b -> Bool
  ```

  In particular, adding constraints such as *HasCallStack* to the instance signature will not have an effect; they need to be added to the class instead.

- One stylistic reason for wanting to write a type signature is simple documentation. Another is that you may want to bring scoped type variables into scope. For example:

```
class C a where
  foo :: b -> a -> (a, [b])

instance C a => C (T a) where
  foo :: forall b. b -> T a -> (T a, [b])
  foo x (T y) = (T y, xs)
     where
       xs :: [b]
       xs = [x,x,x]
```

Provided that you also specify *ScopedTypeVariables* (page 512) (*Lexically scoped type variables* (page 512)), the `forall b` scopes over the definition of `foo`, and in particular over the type signature for `xs`.

6.9 Literals

6.9.1 Negative literals

NegativeLiterals

Since 7.8.1

Enable negative numeric literals.

The literal `-123` is, according to Haskell98 and Haskell 2010, two tokens, a unary minus (`-`) and the number 123, and is desugared as `negate (fromInteger 123)`. The language extension *NegativeLiterals* (page 491) causes it to be treated as a single token and desugared as `fromInteger (-123)`.

This can be useful when the positive and negative range of a numeric data type don't match up. For example, in 8-bit arithmetic -128 is representable, but +128 is not. So `negate (fromInteger 128)` will elicit an unexpected integer-literal-overflow message.

Whitespace can be inserted, as in `- 123`, to force interpretation as two tokens.

In 9.0, the behavior of this extension changed, and now we require that a negative literal must not be preceded by a closing token (see GHC Proposal #229 for the definition of a closing token). In other words, we parse `f -123` as `f (-123)`, but `x-123` as `(-) x 123`. Before this amendment, *NegativeLiterals* (page 491) caused `x-123` to be parsed as `x(-123)`.

NegativeLiterals (page 491) is a subset of *LexicalNegation* (page 317). That is, enabling both of those extensions has the same effect as enabling *LexicalNegation* (page 317) alone.

6.9.2 Binary integer literals

BinaryLiterals

Since 7.10.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow the use of binary notation in integer literals.

Haskell 2010 and Haskell 98 allows for integer literals to be given in decimal, octal (prefixed by `0o` or `0O`), or hexadecimal notation (prefixed by `0x` or `0X`).

The language extension *BinaryLiterals* (page 491) adds support for expressing integer literals in binary notation with the prefix `0b` or `0B`. For instance, the binary integer literal `0b11001001` will be desugared into `fromInteger 201` when *BinaryLiterals* (page 491) is enabled.

6.9.3 Hexadecimal floating point literals

HexFloatLiterals

Since 8.4.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow writing floating point literals using hexadecimal notation.

The hexadecimal notation for floating point literals is useful when you need to specify floating point constants precisely, as the literal notation corresponds closely to the underlying bit-encoding of the number.

In this notation floating point numbers are written using hexadecimal digits, and so the digits are interpreted using base 16, rather then the usual 10. This means that digits left of the decimal point correspond to positive powers of 16, while the ones to the right correspond to negative ones.

You may also write an explicit exponent, which is similar to the exponent in decimal notation with the following differences:

- the exponent begins with `p` instead of `e`
- the exponent is written in base `10` (**not** 16)
- the base of the exponent is `2` (**not** 16).

In terms of the underlying bit encoding, each hexadecimal digit corresponds to 4 bits, and you may think of the exponent as "moving" the floating point by one bit left (negative) or right (positive). Here are some examples:

- `0x0.1` is the same as `1/16`
- `0x0.01` is the same as `1/256`
- `0xF.FF` is the same as `15 + 15/16 + 15/256`
- `0x0.1p4` is the same as `1`
- `0x0.1p-4` is the same as `1/256`
- `0x0.1p12` is the same as `256`

6.9.4 Fractional looking integer literals

NumDecimals

Since 7.8.1

Allow the use of scientific notation style borrowed from floating-point literal syntax for integral types.

Haskell 2010 and Haskell 98 define floating literals with the syntax `1.2e6`, resembling scientific notation. These literals have the type `Fractional a => a`.

The language extension *NumDecimals* (page 492) allows you to also use the scientific notation and floating point literal syntax for instances of `Num`, and have values like `1.2e6 :: Num a => a` and `5e10 :: Num a => a` . This applies only to literals that really turn out to have integral values. For example `1.23e1 :: Fractional a => a` since `1.23e1 == 12.3`, however `1.23e2 :: Num a => a` as `1.23e2 == 123`.

Integral literals written using scientific notation will be desugared using `fromInteger`, whereas any literals which aren't integral will be desugared using `fromRational` as usual.

Note that regular floating point literals (without exponents) will also be desugared via `fromInteger` and assigned type `Num a => a` if they represent an integral value. For example `1.0 :: Num a => a`, but `1.1 :: Fractional a => a`.

6.9.5 Sized primitive literal syntax

ExtendedLiterals

Since 9.8.1

Allows defining unboxed numeric primitive values through `#Type` suffixes on numeric literals e.g. `0xFF#Word8 :: Word8#`.

The *MagicHash* (page 278) extension enables some new literals, including `3# :: Int#`, `3## :: Word#`. This does not extend to all unboxed values. For example, there is no literal syntax for `Word8#`: you must write something such as `wordToWord8 (3## :: Word#) :: Word8#`.

ExtendedLiterals (page 493) enables further syntax for defining primitive numeric literals. Suffix any Haskell integer lexeme with a hash sign `#` followed by a primitive numeric type (without its hash suffix) to obtain a value of that type. For example, `0xFF#Word8 :: Word8#`. There must be no spaces between the parts of the literal.

The primitive numeric types allowed are:

- `Int8#`
- `Int16#`
- `Int32#`
- `Int64#`
- `Int#`
- `Word8#`
- `Word16#`
- `Word32#`
- `Word64#`
- `Word#`

All types permit any nonnegative Haskell integer lexeme, e.g. `70`, `0x2A`, `0o1276`, `0b1010` (with *BinaryLiterals* (page 491)). The signed `Int` types also permit negative integer lexemes. Defining a literal with a value that can't fit in its requested type will emit an overflow warning by default, the same as boxed numeric literals.

As with *MagicHash* (page 278), this extension does not bring anything into scope, nor change any semantics. The syntax only applies to numeric literals. You may want to import `GHC.Exts` (see *Unboxed types and primitive operations* (page 554)) to refer to the types of the literals you define.

6.9.6 Numeric underscores

NumericUnderscores

Since 8.6.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow the use of underscores in numeric literals.

GHC allows for numeric literals to be given in decimal, octal, hexadecimal, binary, or float notation.

The language extension *NumericUnderscores* (page 494) adds support for expressing underscores in numeric literals. For instance, the numeric literal `1_000_000` will be parsed into `1000000` when *NumericUnderscores* (page 494) is enabled. That is, underscores in numeric literals are ignored when *NumericUnderscores* (page 494) is enabled. See also #14473.

For example:

```
-- decimal
million    = 1_000_000
billion    = 1_000_000_000
lightspeed = 299_792_458
version    = 8_04_1
date       = 2017_12_31

-- hexadecimal
red_mask = 0xff_00_00
size1G   = 0x3fff_ffff

-- binary
bit8th   = 0b01_0000_0000
packbits = 0b1_11_01_0000_0_111
bigbits  = 0b1100_1011__1110_1111__0101_0011

-- float
pi       = 3.141_592_653_589_793
faraday  = 96_485.332_89
avogadro = 6.022_140_857e+23

-- function
isUnderMillion = (< 1_000_000)

clip64M x
    | x > 0x3ff_ffff = 0x3ff_ffff
    | otherwise = x

test8bit x = (0b01_0000_0000 .&. x) /= 0
```

About validity:

```
x0 = 1_000_000   -- valid
x1 = 1__000000   -- valid
x2 = 1000000_    -- invalid
x3 = _1000000    -- invalid

e0 = 0.0001      -- valid
e1 = 0.000_1     -- valid
```

(continues on next page)

(continued from previous page)

```
e2 = 0_.0001     -- invalid
e3 = _0.0001     -- invalid
e4 = 0._0001     -- invalid
e5 = 0.0001_     -- invalid

f0 = 1e+23       -- valid
f1 = 1_e+23      -- valid
f2 = 1__e+23     -- valid
f3 = 1e_+23      -- invalid

g0 = 1e+23       -- valid
g1 = 1e+_23      -- invalid
g2 = 1e+23_      -- invalid

h0 = 0xffff      -- valid
h1 = 0xff_ff     -- valid
h2 = 0x_ffff     -- valid
h3 = 0x__ffff    -- valid
h4 = _0xffff     -- invalid
```

6.9.7 Overloaded string literals

OverloadedStrings

Since 6.8.1

Enable overloaded string literals (e.g. string literals desugared via the `IsString` class).

GHC supports *overloaded string literals*. Normally a string literal has type `String`, but with overloaded string literals enabled (with *`OverloadedStrings`* (page 495)) a string literal has type `(IsString a) => a`.

This means that the usual string syntax can be used, e.g., for `ByteString`, `Text`, and other variations of string like types. String literals behave very much like integer literals, i.e., they can be used in both expressions and patterns. If used in a pattern the literal will be replaced by an equality test, in the same way as an integer literal is.

The class `IsString` is defined as:

```
class IsString a where
    fromString :: String -> a
```

The only predefined instance is the obvious one to make strings work as usual:

```
instance IsString [Char] where
    fromString cs = cs
```

The class `IsString` is not in scope by default. If you want to mention it explicitly (for example, to give an instance declaration for it), you can import it from module `Data.String`.

Haskell's defaulting mechanism (Haskell Report, Section 4.3.4) is extended to cover string literals, when *`OverloadedStrings`* (page 495) is specified. Specifically:

- Each type in a `default` declaration must be an instance of `Num` *or* of `IsString`.
- If no `default` declaration is given, then it is just as if the module contained the declaration `default( Integer, Double, String)`.

- The standard defaulting rule is extended thus: defaulting applies when all the unresolved constraints involve standard classes *or* `IsString`; and at least one is a numeric class *or* `IsString`.

So, for example, the expression `length "foo"` will give rise to an ambiguous use of `IsString a0` which, because of the above rules, will default to `String`.

A small example:

```
module Main where

import Data.String( IsString(..) )

newtype MyString = MyString String deriving (Eq, Show)
instance IsString MyString where
    fromString = MyString

greet :: MyString -> MyString
greet "hello" = "world"
greet other = other

main = do
    print $ greet "hello"
    print $ greet "fool"
```

Note that deriving `Eq` is necessary for the pattern matching to work since it gets translated into an equality comparison.

6.9.8 Overloaded labels

`OverloadedLabels`

Since 8.0.1

Enable use of the `#foo` overloaded label syntax.

GHC supports *overloaded labels*, a form of identifier whose interpretation may depend both on its type and on its literal text. When the *`OverloadedLabels`* (page 496) extension is enabled, an overloaded label can be written with a prefix hash, for example `#foo`. The type of this expression is `IsLabel "foo" a => a`.

The class `IsLabel` is defined as:

```
class IsLabel (x :: Symbol) a where
  fromLabel :: a
```

This is rather similar to the class `IsString` (see *Overloaded string literals* (page 495)), but with an additional type parameter that makes the text of the label available as a type-level string (see *Type-Level Literals* (page 384)). Note that `fromLabel` had an extra `Proxy# x` argument in GHC 8.0, but this was removed in GHC 8.2 as a type application (see *Visible type application* (page 386)) can be used instead.

There are no predefined instances of this class. It is not in scope by default, but can be brought into scope by importing GHC.OverloadedLabels. Unlike `IsString`, there are no special defaulting rules for `IsLabel`.

During typechecking, GHC will replace an occurrence of an overloaded label like `#foo` with `fromLabel @"foo"`. This will have some type `alpha` and require the solution of a class constraint `IsLabel "foo" alpha`.

The intention is for `IsLabel` to be used to support overloaded record fields and perhaps anonymous records. Thus, it may be given instances for base datatypes (in particular `(->)`) in the future.

If *RebindableSyntax* (page 295) is enabled, overloaded labels will be desugared using whatever `fromLabel` function is in scope, rather than always using `GHC.OverloadedLabels.fromLabel`.

When writing an overloaded label, there must be no space between the hash sign and the following identifier. The *MagicHash* (page 278) extension makes use of postfix hash signs; if *OverloadedLabels* (page 496) and *MagicHash* (page 278) are both enabled then `x#y` means `x# y`, but if only *OverloadedLabels* (page 496) is enabled then it means `x #y`. The *UnboxedTuples* (page 556) extension makes `(#` a single lexeme, so when *UnboxedTuples* (page 556) is enabled you must write a space between an opening parenthesis and an overloaded label. To avoid confusion, you are strongly encouraged to put a space before the hash when using *OverloadedLabels* (page 496).

When using *OverloadedLabels* (page 496) (or other extensions that make use of hash signs) in a `.hsc` file (see *Writing Haskell interfaces to C code: hsc2hs* (page 699)), the hash signs must be doubled (write `##foo` instead of `#foo`) to avoid them being treated as `hsc2hs` directives.

Here is an extension of the record access example in *Type-Level Literals* (page 384) showing how an overloaded label can be used as a record selector:

```
{-# LANGUAGE DataKinds, KindSignatures, MultiParamTypeClasses,
             FunctionalDependencies, FlexibleInstances,
             OverloadedLabels, ScopedTypeVariables #-}

import GHC.OverloadedLabels (IsLabel(..))
import GHC.TypeLits (Symbol)

data Label (l :: Symbol) = Get

class Has a l b | a l -> b where
  from :: a -> Label l -> b

data Point = Point Int Int deriving Show

instance Has Point "x" Int where from (Point x _) _ = x
instance Has Point "y" Int where from (Point _ y) _ = y

instance Has a l b => IsLabel l (a -> b) where
  fromLabel x = from x (Get :: Label l)

example = #x (Point 1 2)
```

Since GHC 9.6, any non-empty double quoted string can be used as a label. The restriction that the label must be a valid identifier has also been lifted.

Examples of newly allowed syntax:

- Leading capital letters: *#Foo* equivalant to *getLabel @"Foo"*
- Numeric characters: *#3.14* equivalent to *getLabel @"3.14"*
- Arbitrary strings: *#"Hello, World!"* equivalent to *getLabel @"Hello, World!"*

Here is an example of the more permissive use of this extension, available since GHC 9.6:

```
{-# LANGUAGE DataKinds             #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedLabels      #-}
{-# LANGUAGE MagicHash             #-}

import Data.Foldable (traverse_)
import Data.Proxy (Proxy(..))
import GHC.OverloadedLabels (IsLabel(..))
import GHC.TypeLits (KnownSymbol, symbolVal)
import GHC.Prim (Addr#)

instance KnownSymbol symbol => IsLabel symbol String where
  fromLabel = symbolVal (Proxy :: Proxy symbol)

(#) :: String -> Int -> String
(#) _ i = show i

f :: Addr# -> Int -> String
f _ i = show i

main :: IO ()
main = traverse_ putStrLn
  [ #a
  , #number17
  , #do
  , #type
  , #Foo
  , #3
  , #199.4
  , #17a23b
  , #f'a'
  , #'a'
  , #'
  , #''notTHSplice
  , #...
  , #привет
  , #□□□□□
  , #"3"
  , #":"
  , #"Foo"
  , #"The quick brown fox"
  , #"\""
  , (++) #hello#world
  , (++) #"hello"#"world"
  , #"hello"# 1 -- equivalent to `(fromLabel @"hello") # 1`
  , f "hello"#2 -- equivalent to `f ("hello"# :: Addr#) 2`
  ]
```

See GHC Proposal #170 for more details.

6.10 Constraints

6.10.1 Loosening restrictions on class contexts

FlexibleContexts

Since 6.8.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Remove the type-variable restriction on class contexts.

The *FlexibleContexts* (page 499) extension lifts the Haskell 98 restriction that the type-class constraints (anywhere they appear) must have the form *(class type-variable)* or *(class (type-variable type1 type2 … typen))*. With *FlexibleContexts* (page 499) these type signatures are perfectly okay:

```
g :: Eq [a] => ...
g :: Ord (T a ()) => ...
```

This extension does not affect equality constraints in an instance context; they are permitted by *TypeFamilies* (page 338) or *GADTs* (page 335).

Note that *FlexibleContexts* (page 499) affects usages of class constraints, in type signatures and other contexts. In contrast, *FlexibleInstances* (page 480) loosens a similar restriction in place when declaring a new instance.

6.10.2 Equality constraints and Coercible constraint

6.10.2.1 Equality constraints

A type context can include equality constraints of the form `t1 ~ t2`, which denote that the types `t1` and `t2` need to be the same. In the presence of type families, whether two types are equal cannot generally be decided locally. Hence, the contexts of function signatures may include equality constraints, as in the following example:

```
sumCollects :: (Collects c1, Collects c2, Elem c1 ~ Elem c2) => c1 -> c2 -> c2
```

where we require that the element type of `c1` and `c2` are the same. In general, the types `t1` and `t2` of an equality constraint may be arbitrary monotypes; i.e., they may not contain any quantifiers, independent of whether higher-rank types are otherwise enabled.

Equality constraints can also appear in class and instance contexts. The former enable a simple translation of programs using functional dependencies into programs using family synonyms instead. The general idea is to rewrite a class declaration of the form

```
class C a b | a -> b
```

to

```
class (F a ~ b) => C a b where
  type F a
```

That is, we represent every functional dependency (FD) `a1 .. an -> b` by an FD type family `F a1 .. an` and a superclass context equality `F a1 .. an ~ b`, essentially giving a name to the functional dependency. In class instances, we define the type instances of FD families in accordance with the class head. Method signatures are not affected by that process.

6.10.2.2 Heterogeneous equality

GHC also supports *kind-heterogeneous* equality, which relates two types of potentially different kinds. Heterogeneous equality is spelled `~~`. Here are the kinds of `~` and `~~` to better understand their difference:

```
(~)  :: forall k. k -> k -> Constraint
(~~) :: forall k1 k2. k1 -> k2 -> Constraint
```

Users will most likely want `~`, but `~~` is available if GHC cannot know, a priori, that the two types of interest have the same kind. Evidence that `(a :: k1) ~~ (b :: k2)` tells GHC both that `k1` and `k2` are the same and that `a` and `b` are the same.

Because `~` is the more common equality relation, GHC prints out `~~` like `~` unless *-fprint-equality-relations* (page 78) is set.

6.10.2.3 Unlifted heterogeneous equality

Internal to GHC is yet a third equality relation (`~#`). It is heterogeneous (like `~~`) and is used only internally. It may appear in error messages and other output only when *-fprint-equality-relations* (page 78) is enabled.

6.10.2.4 The Coercible constraint

The constraint `Coercible t1 t2` is similar to `t1 ~ t2`, but denotes representational equality between `t1` and `t2` in the sense of Roles (*Roles* (page 417)). It is exported by Data.Coerce, which also contains the documentation. More details and discussion can be found in the paper "Safe Coercions".

6.10.3 The Constraint kind

ConstraintKinds

Since 7.4.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow types of kind `Constraint` to be used in contexts.

Normally, *constraints* (which appear in types to the left of the => arrow) have a very restricted syntax. They can only be:

- Class constraints, e.g. `Show a`
- *Implicit parameter* (page 517) constraints, e.g. `?x::Int` (with the *ImplicitParams* (page 517) extension)
- *Equality constraints* (page 499), e.g. `a ~ Int` (with the *TypeFamilies* (page 338) or *GADTs* (page 335) extensions)

With the *ConstraintKinds* (page 500) extension, GHC becomes more liberal in what it accepts as constraints in your program. To be precise, with this flag any *type* of the new kind `Constraint` can be used as a constraint. The following things have kind `Constraint`:

- Anything which is already valid as a constraint without the flag: saturated applications to type classes, implicit parameter and equality constraints.
- Tuples, all of whose component types have kind `Constraint`. So for example the type `(Show a, Ord a)` is of kind `Constraint`.
- Anything whose form is not yet known, but the user has declared to have kind `Constraint` (for which they need to import it from `Data.Kind`). For example `type Foo (f :: Type -> Constraint) = forall b. f b => b -> b` is allowed.

Note, however, that the *TypeFamilies* (page 338) and *GADTs* (page 335) extensions also allow the manipulation of things with kind `Constraint`, without necessarily requiring the *ConstraintKinds* (page 500) extension:

```
-- With -XTypeFamilies -XNoConstraintKinds
type T :: Type -> (Type -> Constraint)
type family T a where
  T Int    = Num
  T Double = Floating

-- With -XGADTs -XNoConstraintKinds
type Dict :: Constraint -> Type
data Dict c where
  MkDict :: c => Dict c
```

With the *ConstraintKinds* (page 500) extension, constraints are just handled as types of a particular kind. This allows type constraint synonyms:

```
type Stringy a = (Read a, Show a)
foo :: Stringy a => a -> (String, String -> a)
foo x = (show x, read)
```

Presently, only standard constraints, tuples and type synonyms for those two sorts of constraint are permitted in instance contexts and superclasses (without extra flags). The reason is that permitting more general constraints can cause type checking to loop, as it would with these two programs:

```
type family Clsish u a
type instance Clsish () a = Cls a
class Clsish () a => Cls a where
```

```
class OkCls a where

type family OkClsish u a
type instance OkClsish () a = OkCls a
instance OkClsish () a => OkCls a where
```

You may write programs that use exotic sorts of constraints in instance contexts and superclasses, but to do so you must use *UndecidableInstances* (page 484) to signal that you don't mind if the type checker fails to terminate.

6.10.4 Quantified constraints

QuantifiedConstraints

Implies *ExplicitForAll* (page 506)

Since 8.6.1

Allow constraints to quantify over types.

The extension *QuantifiedConstraints* (page 502) introduces **quantified constraints**, which give a new level of expressiveness in constraints. For example, consider

```
data Rose f a = Branch a (f (Rose f a))

instance (Eq a, ???) => Eq (Rose f a)
  where
    (Branch x1 c1) == (Branch x2 c2)
      = x1==x2 && c1==c2
```

From the `x1==x2` we need `Eq a`, which is fine. From `c1==c2` we need `Eq (f (Rose f a))` which is *not* fine in Haskell today; we have no way to solve such a constraint.

QuantifiedConstraints (page 502) lets us write this

```
instance (Eq a, forall b. (Eq b) => Eq (f b))
       => Eq (Rose f a)
  where
    (Branch x1 c1) == (Branch x2 c2)
      = x1==x2 && c1==c2
```

Here, the quantified constraint `forall b. (Eq b) => Eq (f b)` behaves a bit like a local instance declaration, and makes the instance typeable.

The paper Quantified class constraints (by Bottu, Karachalias, Schrijvers, Oliveira, Wadler, Haskell Symposium 2017) describes this feature in technical detail, with examples, and so is a primary reference source for this feature.

6.10.4.1 Motivation

Introducing quantified constraints offers two main benefits:

- Firstly, they enable terminating resolution where this was not possible before. Consider for instance the following instance declaration for the general rose datatype

  ```
  data Rose f x = Rose x (f (Rose f x))

  instance (Eq a, forall b. Eq b => Eq (f b)) => Eq (Rose f a) where
    (Rose x1 rs1) == (Rose x2 rs2) = x1 == x2 && rs1 == rs2
  ```

  This extension allows us to write constraints of the form `forall b. Eq b => Eq (f b)`, which is needed to solve the `Eq (f (Rose f x))` constraint arising from the second usage of the `(==)` method.

- Secondly, quantified constraints allow for more concise and precise specifications. As an example, consider the MTL type class for monad transformers:

  ```
  class Trans t where
    lift :: Monad m => m a -> (t m) a
  ```

The developer knows that a monad transformer takes a monad `m` into a new monad `t m`. But this property is not formally specified in the above declaration. This omission becomes an issue when defining monad transformer composition:

```
newtype (t1 * t2) m a = C { runC :: t1 (t2 m) a }

instance (Trans t1, Trans t2) => Trans (t1 * t2) where
  lift = C . lift . lift
```

The goal here is to `lift` from monad `m` to `t2 m` and then `lift` this again into `t1 (t2 m)`. However, this second `lift` can only be accepted when `(t2 m)` is a monad and there is no way of establishing that this fact universally holds.

Quantified constraints enable this property to be made explicit in the `Trans` class declaration:

```
class (forall m. Monad m => Monad (t m)) => Trans t where
  lift :: Monad m => m a -> (t m) a
```

This idea is very old; see Section 7 of Derivable type classes.

6.10.4.2 Syntax changes

Haskell 2010 defines a `context` (the bit to the left of => in a type) like this

```
context ::= class
        |   ( class1, ..., classn )

class ::= qtycls tyvar
        |  qtycls (tyvar atype1 ... atypen)
```

We extend `class` (warning: this is a rather confusingly named non-terminal symbol) with two extra forms, namely precisely what can appear in an instance declaration

```
class ::= ...
      | [context =>] qtycls inst
      | [context =>] tyvar inst
```

The definition of `inst` is unchanged from the Haskell Report (roughly, just a type). The `context` => part is optional. That is the only syntactic change to the language.

Notes:

- Where GHC allows extensions in instance declarations we allow exactly the same extensions to this new form of `class`. Specifically, with *ExplicitForAll* (page 506) and *MultiParamTypeClasses* (page 470) the syntax becomes

  ```
  class ::= ...
         | [forall tyvars .] [context =>] qtycls inst1 ... instn
         | [forall tyvars .] [context =>] tyvar inst1 ... instn
  ```

  Note that an explicit `forall` is often absolutely essential. Consider the rose-tree example

  ```
  instance (Eq a, forall b. Eq b => Eq (f b)) => Eq (Rose f a) where ...
  ```

  Without the `forall b`, the type variable `b` would be quantified over the whole instance declaration, which is not what is intended.

- One of these new quantified constraints can appear anywhere that any other constraint can, not just in instance declarations. Notably, it can appear in a type signature for a value binding, data constructor, or expression. For example

```
f :: (Eq a, forall b. Eq b => Eq (f b)) => Rose f a -> Rose f a -> Bool
f t1 t2 = not (t1 == t2)
```

- The form with a type variable at the head allows this:

```
instance (forall xx. c (Free c xx)) => Monad (Free c) where
    Free f >>= g = f g
```

  See Iceland Jack's summary. The key point is that the bit to the right of the => may be headed by a type *variable* (c in this case), rather than a class. It should not be one of the forall'd variables, though.

  (NB: this goes beyond what is described in the paper, but does not seem to introduce any new technical difficulties.)

6.10.4.3 Typing changes

See the paper.

6.10.4.4 Superclasses

Suppose we have:

```
f :: forall m. (forall a. Ord a => Ord (m a)) => m Int -> Bool
f x = x == x
```

From the x==x we need an Eq (m Int) constraint, but the context only gives us a way to figure out Ord (m a) constraints. But from the given constraint forall a. Ord a => Ord (m a) we derive a second given constraint forall a. Ord a => Eq (m a), and from that we can readily solve Eq (m Int). This process is very similar to the way that superclasses already work: given an Ord a constraint we derive a second given Eq a constraint.

NB: This treatment of superclasses goes beyond the paper, but is specifically desired by users.

6.10.4.5 Overlap

Quantified constraints can potentially lead to overlapping local axioms. Consider for instance the following example:

```
class A a where {}
class B a where {}
class C a where {}
class (A a => C a) => D a where {}
class (B a => C a) => E a where {}

class C a => F a where {}
instance (B a, D a, E a) => F a where {}
```

When type checking the instance declaration for F a, we need to check that the superclass C of F holds. We thus try to entail the constraint C a under the theory containing:

- The instance axioms : `(B a, D a, E a) => F a`
- The local axioms from the instance context : `B a`, `D a` and `E a`
- The closure of the superclass relation over these local axioms : `A a => C a` and `B a => C a`

However, the `A a => C a` and `B a => C a` axioms both match the wanted constraint `C a`. There are several possible approaches for handling these overlapping local axioms:

- **Pick first**. We can simply select the **first matching axiom** we encounter. In the above example, this would be `A a => C a`. We'd then need to entail `A a`, for which we have no matching axioms available, causing the above program to be rejected.

  But suppose we made a slight adjustment to the order of the instance context, putting `E a` before `D a`:

  ```
  instance (B a, E a, D a) => F a where {}
  ```

  The first matching axiom we encounter while entailing `C a`, is `B a => C a`. We have a local axiom `B a` available, so now the program is suddenly accepted. This behaviour, where the ordering of an instance context determines whether or not the program is accepted, seems rather confusing for the developer.

- **Reject if in doubt**. An alternative approach would be to check for overlapping axioms, when solving a constraint. When multiple matching axioms are discovered, we **reject the program**. This approach is a bit conservative, in that it may reject working programs. But it seem much more transparent towards the developer, who can be presented with a clear message, explaining why the program is rejected.

- **Backtracking**. Lastly, a simple form of **backtracking** could be introduced. We simply select the first matching axiom we encounter and when the entailment fails, we backtrack and look for other axioms that might match the wanted constraint.

  This seems the most intuitive and transparent approach towards the developer, who no longer needs to concern himself with the fact that his code might contain overlapping axioms or with the ordering of his instance contexts. But backtracking would apply equally to ordinary instance selection (in the presence of overlapping instances), so it is a much more pervasive change, with substantial consequences for the type inference engine.

GHC adopts **Reject if in doubt** for now. We can see how painful it is in practice, and try something more ambitious if necessary.

6.10.4.6 Instance lookup

In the light of the overlap decision, instance lookup works like this when trying to solve a class constraint `C t`

1. First see if there is a given un-quantified constraint `C t`. If so, use it to solve the constraint.
2. If not, look at all the available given quantified constraints; if exactly one matches `C t`, choose it; if more than one matches, report an error.
3. If no quantified constraints match, look up in the global instances, as described in *Instance declarations and resolution* (page 480) and *Overlapping instances* (page 486).

6.10.4.7 Termination

GHC uses the *Paterson Conditions* (page 483) to ensure that instance resolution terminates. How are those rules modified for quantified constraints? In two ways.

- Each quantified constraint, taken by itself, must satisfy the termination rules for an instance declaration.
- After "for each class constraint `(C t1 ... tn)`", add "or each quantified constraint `(forall as. context => C t1 .. tn)`"

Note that the second item only at the *head* of the quantified constraint, not its context. Reason: the head is the new goal that has to be solved if we use the instance declaration.

Of course, `UndecidableInstances` lifts the Paterson Conditions, as now.

6.10.4.8 Coherence

Although quantified constraints are a little like local instance declarations, they differ in one big way: the local instances are written by the compiler, not the user, and hence cannot introduce incoherence. Consider

```
f :: (forall a. Eq a => Eq (f a)) => f b -> f Bool
f x = ...rhs...
```

In `...rhs...` there is, in effect a local instance for `Eq (f a)` for any `a`. But at a call site for `f` the compiler itself produces evidence to pass to `f`. For example, if we called `f Nothing`, then `f` is `Maybe` and the compiler must prove (at the call site) that `forall a. Eq a => Eq (Maybe a)` holds. It can do this easily, by appealing to the existing instance declaration for `Eq (Maybe a)`.

In short, quantified constraints do not introduce incoherence.

6.11 Type signatures

6.11.1 Explicit universal quantification (forall)

`ExplicitForAll`

> **Implied by** *`ScopedTypeVariables`* (page 512), *`LiberalTypeSynonyms`* (page 324), *`RankNTypes`* (page 402), *`ExistentialQuantification`* (page 325)
>
> **Since** 6.12.1
>
> **Status** Included in *`GHC2024`* (page 270), *`GHC2021`* (page 271)

Allow use of the `forall` keyword in places where universal quantification is implicit.

Haskell type signatures are implicitly quantified. When the language option *`ExplicitForAll`* (page 506) is used, the keyword `forall` allows us to say exactly what this means. For example:

```
g :: b -> b
```

means this:

```
g :: forall b. (b -> b)
```

The two are treated identically, except that the latter may bring type variables into scope (see *Lexically scoped type variables* (page 512)).

This extension also enables explicit quantification of type and kind variables in *Data instance declarations* (page 340), *Type instance declarations* (page 343), *Closed type families* (page 344), *Associated instances* (page 348), and *Rewrite rules* (page 589).

Notes:

- As well as in type signatures, you can also use an explicit `forall` in an instance declaration:

  ```
  instance forall a. Eq a => Eq [a] where ...
  ```

  Note that the use of `forall`s in instance declarations is somewhat restricted in comparison to other types. For example, instance declarations are not allowed to contain nested `forall`s. See *Formal syntax for instance declaration types* (page 481) for more information.

- If the `-Wunused-foralls` (page 104) flag is enabled, a warning will be emitted when you write a type variable in an explicit `forall` statement that is otherwise unused. For instance:

  ```
  g :: forall a b. (b -> b)
  ```

  would warn about the unused type variable *a*.

6.11.1.1 The `forall`-or-nothing rule

In certain forms of types, type variables obey what is known as the "`forall`-or-nothing" rule: if a type has an outermost, explicit, invisible `forall`, then all of the type variables in the type must be explicitly quantified. These two examples illustrate how the rule works:

```
f  :: forall a b. a -> b -> b         -- OK, `a` and `b` are explicitly bound
g  :: forall a. a -> forall b. b -> b -- OK, `a` and `b` are explicitly bound
h  :: forall a. a -> b -> b           -- Rejected, `b` is not in scope
```

The type signatures for `f`, `g`, and `h` all begin with an outermost invisible `forall`, so every type variable in these signatures must be explicitly bound by a `forall`. Both `f` and `g` obey the `forall`-or-nothing rule, since they explicitly quantify `a` and `b`. On the other hand, `h` does not explicitly quantify `b`, so GHC will reject its type signature for being improperly scoped.

In places where the `forall`-or-nothing rule takes effect, if a type does *not* have an outermost invisible `forall`, then any type variables that are not explicitly bound by a `forall` become implicitly quantified. For example:

```
i :: a -> b -> b             -- `a` and `b` are implicitly quantified
j :: a -> forall b. b -> b   -- `a` is implicitly quantified
k :: (forall a. a -> b -> b) -- `b` is implicitly quantified
type L :: forall a -> b -> b -- `b` is implicitly quantified
```

GHC will accept `i`, `j`, and `k`'s type signatures, as well as `L`'s kind signature. Note that:

- `j`'s signature is accepted despite its mixture of implicit and explicit quantification. As long as a `forall` is not an outermost one, it is fine to use it among implicitly bound type variables.

- `k`'s signature is accepted because the outermost parentheses imply that the `forall` is not an outermost `forall`. The `forall`-or-nothing rule is one of the few places in GHC where the presence or absence of parentheses can be semantically significant!
- `L`'s signature begins with an outermost `forall`, but it is a *visible* `forall`, not an invisible `forall`, and therefore does not trigger the `forall`-or-nothing rule.

The `forall`-or-nothing rule takes effect in the following places:

- Type signature declarations for functions, values, and class methods
- Expression type annotations
- Instance declarations
- *Default method signatures* (page 472)
- Type signatures in a *SPECIALIZE pragma* (page 614) or *SPECIALIZE instance pragma* (page 617)
- *Standalone kind signatures and polymorphic recursion* (page 369)
- Type signatures for *Generalised Algebraic Data Types (GADTs)* (page 335) constructors
- Type signatures for *Pattern synonyms* (page 461)
- *Data instance declarations* (page 340), *Type instance declarations* (page 343), *Closed type families* (page 344), and *Associated instances* (page 348)
- *Rewrite rules* (page 589) in which the type variables are explicitly quantified

Notes:

- *Pattern type signatures* (page 514) are a notable example of a place where types do *not* obey the `forall`-or-nothing rule. For example, GHC will accept the following:

  ```
  f (g :: forall a. a -> b) x = g x :: b
  ```

  Furthermore, *Rewrite rules* (page 589) do not obey the `forall`-or-nothing rule when their type variables are not explicitly quantified:

  ```
  {-# RULES "f" forall (g :: forall a. a -> b) x. f g x = g x :: b #-}
  ```

- GADT constructors are extra particular about their `forall`s. In addition to adhering to the `forall`-or-nothing rule, GADT constructors also forbid nested `forall`s. For example, GHC would reject the following GADT:

  ```
  data T where
    MkT :: (forall a. a -> b -> T)
  ```

  Because of the lack of an outermost `forall` in the type of `MkT`, the `b` would be implicitly quantified. In effect, it would be as if one had written `MkT :: forall b. (forall a. a -> b -> T)`, which contains nested `forall`s. See *Formal syntax for GADTs* (page 330).

6.11.2 Ambiguous types and the ambiguity check

AllowAmbiguousTypes

Since 7.8.1

Allow type signatures which appear that they would result in an unusable binding.

Each user-written type signature is subjected to an *ambiguity check*. The ambiguity check rejects functions that can never be called. For example:

```
f :: C a => Int
```

The idea is there can be no legal calls to `f` because every call will give rise to an ambiguous constraint. Indeed, the *only* purpose of the ambiguity check is to report functions that cannot possibly be called. We could soundly omit the ambiguity check on type signatures entirely, at the expense of delaying ambiguity errors to call sites. Indeed, the language extension *AllowAmbiguousTypes* (page 509) switches off the ambiguity check.

Ambiguity can be subtle. Consider this example which uses functional dependencies:

```
class D a b | a -> b where ..
h :: D Int b => Int
```

The `Int` may well fix `b` at the call site, so that signature should not be rejected. Moreover, the dependencies might be hidden. Consider

```
class X a b where ...
class D a b | a -> b where ...
instance D a b => X [a] b where...
h :: X a b => a -> a
```

Here `h`'s type looks ambiguous in `b`, but here's a legal call:

```
...(h [True])...
```

That gives rise to a `(X [Bool] beta)` constraint, and using the instance means we need `(D Bool beta)` and that fixes `beta` via `D`'s fundep!

Behind all these special cases there is a simple guiding principle. Consider

```
f :: type
f = ...blah...

g :: type
g = f
```

You would think that the definition of `g` would surely typecheck! After all `f` has exactly the same type, and `g=f`. But in fact `f`'s type is instantiated and the instantiated constraints are solved against the constraints bound by `g`'s signature. So, in the case an ambiguous type, solving will fail. For example, consider the earlier definition `f :: C a => Int`:

```
f :: C a => Int
f = ...blah...

g :: C a => Int
g = f
```

In g's definition, we'll instantiate to (C alpha) and try to deduce (C alpha) from (C a), and fail.

So in fact we use this as our *definition* of ambiguity: a type ty is ambiguous if and only if ((undefined :: ty) :: ty) would fail to typecheck. We use a very similar test for *inferred* types, to ensure that they too are unambiguous.

Switching off the ambiguity check. Even if a function has an ambiguous type according to the "guiding principle", it is possible that the function is callable. For example:

```
class D a b where ...
instance D Bool b where ...

strange :: D a b => a -> a
strange = ...blah...

foo = strange True
```

Here strange's type is ambiguous, but the call in foo is OK because it gives rise to a constraint (D Bool beta), which is soluble by the (D Bool b) instance.

Another way of getting rid of the ambiguity at the call site is to use the *TypeApplications* (page 386) extension to specify the types. For example:

```
class D a b where
  h :: b
instance D Int Int where ...

main = print (h @Int @Int)
```

Here a is ambiguous in the definition of D but later specified to be *Int* using type applications.

AllowAmbiguousTypes (page 509) allows you to switch off the ambiguity check altogether.

Sometimes *AllowAmbiguousTypes* (page 509) does not mix well with *RankNTypes* (page 402). For example:

```
foo :: forall r. (forall i. (KnownNat i) => r) -> r
foo f = f @1

boo :: forall j. (KnownNat j) => Int
boo = ....

h :: Int
h = foo boo
```

This program will be rejected as ambiguous because GHC will not unify the type variables *j* and *i*.

Unlike the previous examples, it is not currently possible to resolve the ambiguity manually by using *TypeApplications* (page 386).

Note: *A historical note.* GHC used to impose some more restrictive and less principled conditions on type signatures. For type forall tv1..tvn (c1, ...,cn) => type GHC used to require

a. that each universally quantified type variable tvi must be "reachable" from type, and
b. that every constraint ci mentions at least one of the universally quantified type variables tvi. These ad-hoc restrictions are completely subsumed by the new ambiguity check.

6.11.3 Explicitly-kinded quantification

KindSignatures

Implied by *TypeFamilies* (page 338), *PolyKinds* (page 364)

Since 6.8.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow explicit kind signatures on type variables.

Haskell infers the kind of each type variable. Sometimes it is nice to be able to give the kind explicitly as (machine-checked) documentation, just as it is nice to give a type signature for a function. On some occasions, it is essential to do so. For example, in his paper "Restricted Data Types in Haskell" (Haskell Workshop 1999) John Hughes had to define the data type:

```
data Set cxt a = Set [a]
               | Unused (cxt a -> ())
```

The only use for the `Unused` constructor was to force the correct kind for the type variable `cxt`.

GHC now instead allows you to specify the kind of a type variable directly, wherever a type variable is explicitly bound, with the extension *KindSignatures* (page 511).

This extension enables kind signatures in the following places:

- `data` declarations:

```
data Set (cxt :: Type -> Type) a = Set [a]
```

- `newtype` declarations:

```
newtype Set (cxt :: Type -> Type) a = Set [a]
```

- `type` declarations:

```
type T (f :: Type -> Type) = f Int
```

- `class` declarations:

```
class (Eq a) => C (f :: Type -> Type) a where ...
```

- `forall`'s in type signatures:

```
f :: forall (cxt :: Type -> Type). Set cxt Int
```

The parentheses are required.

As part of the same extension, you can put kind annotations in types as well. Thus:

```
f :: (Int :: Type) -> Int
g :: forall a. a -> (a :: Type)
```

The syntax is

```
atype ::= '(' ctype '::' kind ')
```

The parentheses are required.

6.11.4 Lexically scoped type variables

ScopedTypeVariables

Implies *ExplicitForAll* (page 506)

Since 6.8.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Enable lexical scoping of type variables explicitly introduced with `forall`.

Tip: *ScopedTypeVariables* (page 512) breaks GHC's usual rule that explicit `forall` is optional and doesn't affect semantics. For the *Declaration type signatures* (page 513) (or *Expression type signatures* (page 514)) examples in this section, the explicit `forall` is required. (If omitted, usually the program will not compile; in a few cases it will compile but the functions get a different signature.) To trigger those forms of *ScopedTypeVariables* (page 512), the `forall` must appear against the top-level signature (or outer expression) but *not* against nested signatures referring to the same type variables.

Explicit `forall` is not always required – see *pattern signature equivalent* (page 512) for the example in this section, or *Pattern type signatures* (page 514).

GHC supports *lexically scoped type variables*, without which some type signatures are simply impossible to write. For example:

```
f :: forall a. [a] -> [a]
f xs = ys ++ ys
     where
       ys :: [a]
       ys = reverse xs
```

The type signature for `f` brings the type variable `a` into scope, because of the explicit `forall` (*Declaration type signatures* (page 513)). The type variables bound by a `forall` scope over the entire definition of the accompanying value declaration. In this example, the type variable `a` scopes over the whole definition of `f`, including over the type signature for `ys`. In Haskell 98 it is not possible to declare a type for `ys`; a major benefit of scoped type variables is that it becomes possible to do so.

An equivalent form for that example, avoiding explicit `forall` uses *Pattern type signatures* (page 514):

```
f :: [a] -> [a]
f (xs :: [aa]) = xs ++ ys
  where
    ys :: [aa]
    ys = reverse xs
```

Unlike the `forall` form, type variable `a` from `f`'s signature is not scoped over `f`'s equation(s). Type variable `aa` bound by the pattern signature is scoped over the right-hand side of `f`'s equation. (Therefore there is no need to use a distinct type variable; using `a` would be equivalent.)

6.11.4.1 Overview

The design follows the following principles

- A scoped type variable stands for a type *variable*, and not for a *type*. (This is a change from GHC's earlier design.)
- Furthermore, distinct lexical type variables stand for distinct type variables. This means that every programmer-written type signature (including one that contains free scoped type variables) denotes a *rigid* type; that is, the type is fully known to the type checker, and no inference is involved.
- Lexical type variables may be alpha-renamed freely, without changing the program.

A *lexically scoped type variable* can be bound by:

- A declaration type signature (*Declaration type signatures* (page 513))
- An expression type signature (*Expression type signatures* (page 514))
- A pattern type signature (*Pattern type signatures* (page 514))
- Class and instance declarations (*Class and instance declarations* (page 516))

In Haskell, a programmer-written type signature is implicitly quantified over its free type variables (Section 4.1.2 of the Haskell Report). Lexically scoped type variables affect this implicit quantification rules as follows: any type variable that is in scope is *not* universally quantified. For example, if type variable `a` is in scope, then

```
(e :: a -> a)     means     (e :: a -> a)
(e :: b -> b)     means     (e :: forall b. b->b)
(e :: a -> b)     means     (e :: forall b. a->b)
```

6.11.4.2 Declaration type signatures

A declaration type signature that has *explicit* quantification (using `forall`) brings into scope the explicitly-quantified type variables, in the definition of the named function. For example:

```
f :: forall a. [a] -> [a]
f (x:xs) = xs ++ [ x :: a ]
```

The "`forall a`" brings "`a`" into scope in the definition of "`f`".

This only happens if:

- The quantification in `f`'s type signature is explicit. For example:

  ```
  g :: [a] -> [a]
  g (x:xs) = xs ++ [ x :: a ]
  ```

  This program will be rejected, because "`a`" does not scope over the definition of "`g`", so "`x::a`" means "`x::forall a. a`" by Haskell's usual implicit quantification rules.

- The type variable is quantified by the single, syntactically visible, outermost `forall` of the type signature. For example, GHC will reject all of the following examples:

  ```
  f1 :: forall a. forall b. a -> [b] -> [b]
  f1 _ (x:xs) = xs ++ [ x :: b ]
  ```

(continues on next page)

(continued from previous page)

```
f2 :: forall a. a -> forall b. [b] -> [b]
f2 _ (x:xs) = xs ++ [ x :: b ]

type Foo = forall b. [b] -> [b]

f3 :: Foo
f3 (x:xs) = xs ++ [ x :: b ]
```

In `f1` and `f2`, the type variable `b` is not quantified by the outermost `forall`, so it is not in scope over the bodies of the functions. Neither is `b` in scope over the body of `f3`, as the `forall` is tucked underneath the `Foo` type synonym.

- The signature gives a type for a function binding or a bare variable binding, not a pattern binding. For example:

```
f1 :: forall a. [a] -> [a]
f1 (x:xs) = xs ++ [ x :: a ]   -- OK

f2 :: forall a. [a] -> [a]
f2 = \(x:xs) -> xs ++ [ x :: a ]   -- OK

f3 :: forall a. [a] -> [a]
Just f3 = Just (\(x:xs) -> xs ++ [ x :: a ])   -- Not OK!
```

`f1` is a function binding, and `f2` binds a bare variable; in both cases the type signature brings `a` into scope. However the binding for `f3` is a pattern binding, and so `f3` is a fresh variable brought into scope by the pattern, not connected with top level `f3`. Then type variable `a` is not in scope of the right-hand side of `Just f3 = ...`.

6.11.4.3 Expression type signatures

An expression type signature that has *explicit* quantification (using `forall`) brings into scope the explicitly-quantified type variables, in the annotated expression. For example:

```
f = runST ( (op >>= \(x :: STRef s Int) -> g x) :: forall s. ST s Bool )
```

Here, the type signature `forall s. ST s Bool` brings the type variable `s` into scope, in the annotated expression `(op >>= \(x :: STRef s Int) -> g x)`.

6.11.4.4 Pattern type signatures

A type signature may occur in any pattern; this is a *pattern type signature*. For example:

```
-- f and g assume that 'a' is already in scope
f = \(x::Int, y::a) -> x

g (x::a) = x

h ((x,y) :: (Int,Bool)) = (y,x)
```

In the case where all the type variables in the pattern type signature are already in scope (i.e. bound by the enclosing context), matters are simple: the signature simply constrains the type of the pattern in the obvious way.

Unlike expression and declaration type signatures, pattern type signatures are not implicitly generalised. The pattern in a *pattern binding* may only mention type variables that are already in scope. For example:

```
f :: forall a. [a] -> (Int, [a])
f xs = (n, zs)
  where
    (ys::[a], n) = (reverse xs, length xs) -- OK
    (zs::[a])    = xs ++ ys                     -- OK

    Just (v::b)  = ...  -- Not OK; b is not in scope
```

Here, the pattern signatures for `ys` and `zs` are fine, but the one for `v` is not because `b` is not in scope.

However, in all patterns *other* than pattern bindings, a pattern type signature may mention a type variable that is not in scope; in this case, *the signature brings that type variable into scope*. For example:

```
-- same f and g as above, now assuming that 'a' is not already in scope
f = \(x::Int, y::a) -> x           -- 'a' is in scope on RHS of ->

g (x::a) = x :: a

hh (Just (v :: b)) = v :: b
```

The pattern type signature makes the type variable available on the right-hand side of the equation.

Bringing type variables into scope is particularly important for existential data constructors. For example:

```
data T = forall a. MkT [a]

k :: T -> T
k (MkT [t::a]) =
    MkT t3
  where
    (t3::[a]) = [t,t,t]
```

Here, the pattern type signature `[t::a]` mentions a lexical type variable that is not already in scope. Indeed, it *must not* already be in scope, because it is bound by the pattern match. The effect is to bring it into scope, standing for the existentially-bound type variable.

It does seem odd that the existentially-bound type variable *must not* be already in scope. Contrast that usually name-bindings merely shadow (make a 'hole') in a same-named outer variable's scope. But we must have *some* way to bring such type variables into scope, else we could not name existentially-bound type variables in subsequent type signatures.

Compare the two (identical) definitions for examples `f`, `g`; they are both legal whether or not `a` is already in scope. They differ in that *if* `a` is already in scope, the signature constrains the pattern, rather than the pattern binding the variable.

6.11.4.5 Class and instance declarations

ScopedTypeVariables (page 512) allow the type variables bound by the top of a `class` or `instance` declaration to scope over the methods defined in the `where` part. Unlike *Declaration type signatures* (page 513), type variables from class and instance declarations can be lexically scoped without an explicit `forall` (although you are allowed an explicit `forall` in an `instance` declaration; see *Explicit universal quantification (forall)* (page 506)). For example:

```
class C a where
  op :: [a] -> a

  op xs = let ys::[a]
              ys = reverse xs
          in
          head ys

instance C b => C [b] where
  op xs = reverse (head (xs :: [[b]]))

-- Alternatively, one could write the instance above as:
instance forall b. C b => C [b] where
  op xs = reverse (head (xs :: [[b]]))
```

While *ScopedTypeVariables* (page 512) is required for type variables from the top of a class or instance declaration to scope over the /bodies/ of the methods, it is not required for the type variables to scope over the /type signatures/ of the methods. For example, the following will be accepted without explicitly enabling *ScopedTypeVariables* (page 512):

```
class D a where
  m :: a -> a

instance Num a => D [a] where
  m :: [a] -> [a]
  m x = map (*2) x
```

Note that writing `m :: [a] -> [a]` requires the use of the *InstanceSigs* (page 490) extension.

Similarly, *ScopedTypeVariables* (page 512) is not required for type variables from the top of the class or instance declaration to scope over associated type families, which only requires the *TypeFamilies* (page 338) extension. For instance, the following will be accepted without explicitly enabling *ScopedTypeVariables* (page 512):

```
class E a where
  type T a

instance E [a] where
  type T [a] = a
```

See *Scoping of class parameters* (page 351) for further information.

6.11.5 Implicit parameters

ImplicitParams

Since 6.8.1

Allow definition of functions expecting implicit parameters.

Implicit parameters are implemented as described in [Lewis2000] and enabled with the option *ImplicitParams* (page 517). (Most of the following, still rather incomplete, documentation is due to Jeff Lewis.)

A variable is called *dynamically bound* when it is bound by the calling context of a function and *statically bound* when bound by the callee's context. In Haskell, all variables are statically bound. Dynamic binding of variables is a notion that goes back to Lisp, but was later discarded in more modern incarnations, such as Scheme. Dynamic binding can be very confusing in an untyped language, and unfortunately, typed languages, in particular Hindley-Milner typed languages like Haskell, only support static scoping of variables.

However, by a simple extension to the type class system of Haskell, we can support dynamic binding. Basically, we express the use of a dynamically bound variable as a constraint on the type. These constraints lead to types of the form `(?x::t') => t`, which says "this function uses a dynamically-bound variable `?x` of type `t'`". For example, the following expresses the type of a sort function, implicitly parameterised by a comparison function named `cmp`.

```
sort :: (?cmp :: a -> a -> Bool) => [a] -> [a]
```

The dynamic binding constraints are just a new form of predicate in the type class system.

An implicit parameter occurs in an expression using the special form `?x`, where `x` is any valid identifier (e.g. `ord ?x` is a valid expression). Use of this construct also introduces a new dynamic-binding constraint in the type of the expression. For example, the following definition shows how we can define an implicitly parameterised sort function in terms of an explicitly parameterised `sortBy` function:

```
sortBy :: (a -> a -> Bool) -> [a] -> [a]

sort   :: (?cmp :: a -> a -> Bool) => [a] -> [a]
sort    = sortBy ?cmp
```

6.11.5.1 Implicit-parameter type constraints

Dynamic binding constraints behave just like other type class constraints in that they are automatically propagated. Thus, when a function is used, its implicit parameters are inherited by the function that called it. For example, our `sort` function might be used to pick out the least value in a list:

```
least   :: (?cmp :: a -> a -> Bool) => [a] -> a
least xs = head (sort xs)
```

Without lifting a finger, the `?cmp` parameter is propagated to become a parameter of `least` as well. With explicit parameters, the default is that parameters must always be explicit propagated. With implicit parameters, the default is to always propagate them.

An implicit-parameter type constraint differs from other type class constraints in the following way: All uses of a particular implicit parameter must have the same type. This means that the type of `(?x, ?x)` is `(?x::a) => (a,a)`, and not `(?x::a, ?x::b) => (a, b)`, as would be the case for type class constraints.

You can't have an implicit parameter in the context of a class or instance declaration. For example, both these declarations are illegal:

```
class (?x::Int) => C a where ...
instance (?x::a) => Foo [a] where ...
```

Reason: exactly which implicit parameter you pick up depends on exactly where you invoke a function. But the "invocation" of instance declarations is done behind the scenes by the compiler, so it's hard to figure out exactly where it is done. Easiest thing is to outlaw the offending types.

Implicit-parameter constraints do not cause ambiguity. For example, consider:

```
f :: (?x :: [a]) => Int -> Int
f n = n + length ?x

g :: (Read a, Show a) => String -> String
g s = show (read s)
```

Here, `g` has an ambiguous type, and is rejected, but `f` is fine. The binding for `?x` at `f`'s call site is quite unambiguous, and fixes the type `a`.

6.11.5.2 Implicit-parameter bindings

An implicit parameter is *bound* using the standard `let` or `where` binding forms. For example, we define the `min` function by binding `cmp`.

```
min :: Ord a => [a] -> a
min  = let ?cmp = (<=) in least
```

A group of implicit-parameter bindings may occur anywhere a normal group of Haskell bindings can occur, except at top level. That is, they can occur in a `let` (including in a list comprehension, or do-notation, or pattern guards), or a `where` clause. Note the following points:

- An implicit-parameter binding group must be a collection of simple bindings to implicit-style variables (no function-style bindings, and no type signatures); these bindings are neither polymorphic or recursive.
- You may not mix implicit-parameter bindings with ordinary bindings in a single `let` expression; use two nested `let`s instead. (In the case of `where` you are stuck, since you can't nest `where` clauses.)
- You may put multiple implicit-parameter bindings in a single binding group; but they are *not* treated as a mutually recursive group (as ordinary `let` bindings are). Instead they are treated as a non-recursive group, simultaneously binding all the implicit parameter. The bindings are not nested, and may be re-ordered without changing the meaning of the program. For example, consider:

  ```
  f t = let { ?x = t; ?y = ?x+(1::Int) } in ?x + ?y
  ```

  The use of `?x` in the binding for `?y` does not "see" the binding for `?x`, so the type of `f` is

  ```
  f :: (?x::Int) => Int -> Int
  ```

6.11.5.3 Implicit parameters and polymorphic recursion

Consider these two definitions:

```
len1 :: [a] -> Int
len1 xs = let ?acc = 0 in len_acc1 xs

len_acc1 [] = ?acc
len_acc1 (x:xs) = let ?acc = ?acc + (1::Int) in len_acc1 xs

------------

len2 :: [a] -> Int
len2 xs = let ?acc = 0 in len_acc2 xs

len_acc2 :: (?acc :: Int) => [a] -> Int
len_acc2 [] = ?acc
len_acc2 (x:xs) = let ?acc = ?acc + (1::Int) in len_acc2 xs
```

The only difference between the two groups is that in the second group `len_acc` is given a type signature. In the former case, `len_acc1` is monomorphic in its own right-hand side, so the implicit parameter `?acc` is not passed to the recursive call. In the latter case, because `len_acc2` has a type signature, the recursive call is made to the *polymorphic* version, which takes `?acc` as an implicit parameter. So we get the following results in GHCi:

```
Prog> len1 "hello"
0
Prog> len2 "hello"
5
```

Adding a type signature dramatically changes the result! This is a rather counter-intuitive phenomenon, worth watching out for.

6.11.5.4 Implicit parameters scoping guarantees

GHC always takes the most nested implicit parameter binding from the context to find the value. Consider the following code:

```
let ?f = 1 in let ?f = 2 in ?f
```

This expression will always return 2.

Another example of this rule is matching over constructors with constraints. For example:

```
data T where
  MkT :: (?f :: Int) => T

f :: T -> T -> Int
f MkT MkT = ?f
```

Here GHC will always take `?f` from the last match.

6.11.5.5 Implicit parameters and monomorphism

GHC applies the dreaded Monomorphism Restriction (section 4.5.5 of the Haskell Report) to implicit parameters. For example, consider:

```
f :: Int -> Int
f v = let ?x = 0     in
      let y = ?x + v in
      let ?x = 5     in
      y
```

Since the binding for `y` falls under the Monomorphism Restriction it is not generalised, so the type of `y` is simply `Int`, not `(?x::Int) => Int`. Hence, `(f 9)` returns result `9`. If you add a type signature for `y`, then `y` will get type `(?x::Int) => Int`, so the occurrence of `y` in the body of the `let` will see the inner binding of `?x`, so `(f 9)` will return `14`.

6.11.6 Partial Type Signatures

PartialTypeSignatures

Since 7.10.1

Type checker will allow inferred types for holes.

A partial type signature is a type signature containing special placeholders called *wildcards*. A wildcard is written as an underscore (e.g. "`_`") or, if *NamedWildCards* (page 522) is enabled, any identifier with a leading underscore (e.g. "`_foo`", "`_bar`"). Partial type signatures are to type signatures what *Typed Holes* (page 304) are to expressions. During compilation these wildcards or holes will generate an error message that describes which type was inferred at the hole's location, and information about the origin of any free type variables. GHC reports such error messages by default.

Unlike *Typed Holes* (page 304), which make the program incomplete and will generate errors when they are evaluated, this needn't be the case for holes in type signatures. The type checker is capable (in most cases) of type-checking a binding with or without a type signature. A partial type signature bridges the gap between the two extremes, the programmer can choose which parts of a type to annotate and which to leave over to the type-checker to infer.

By default, the type-checker will report an error message for each hole in a partial type signature, informing the programmer of the inferred type. When the *PartialTypeSignatures* (page 520) extension is enabled, the type-checker will accept the inferred type for each hole, generating warnings instead of errors. Additionally, these warnings can be silenced with the *-Wno-partial-type-signatures* (page 88) flag.

However, because GHC must *infer* the type when part of a type is left out, it is unable to use polymorphic recursion. The same restriction takes place when the type signature is omitted completely.

A partial type signature als makes GHC generalise the binding even if *MonoLocalBinds* (page 526) is on; see *Let-generalisation* (page 526).

6.11.6.1 Syntax

A (partial) type signature has the following form: `forall a b .. . (C1, C2, ..) => tau`. It consists of three parts:

- The type variables: `a b ..`
- The constraints: `(C1, C2, ..)`
- The (mono)type: `tau`

We distinguish three kinds of wildcards.

Type Wildcards

Wildcards occurring within the monotype (tau) part of the type signature are *type wildcards* ("type" is often omitted as this is the default kind of wildcard). Type wildcards can be instantiated to any monotype like `Bool` or `Maybe [Bool]`, including functions and higher-kinded types like `(Int -> Bool)` or `Maybe`.

```
not' :: Bool -> _
not' x = not x
-- Inferred: Bool -> Bool

maybools :: _
maybools = Just [True]
-- Inferred: Maybe [Bool]

just1 :: _ Int
just1 = Just 1
-- Inferred: Maybe Int

filterInt :: _ -> _ -> [Int]
filterInt = filter -- has type forall a. (a -> Bool) -> [a] -> [a]
-- Inferred: (Int -> Bool) -> [Int] -> [Int]
```

For instance, the first wildcard in the type signature `not'` would produce the following error message:

```
Test.hs:4:17: error:
    • Found type wildcard ‘_’ standing for ‘Bool’
      To use the inferred type, enable PartialTypeSignatures
    • In the type signature:
        not' :: Bool -> _
    • Relevant bindings include
        not' :: Bool -> Bool (bound at Test.hs:5:1)
```

When a wildcard is not instantiated to a monotype, it will be generalised over, i.e. replaced by a fresh type variable, e.g.

```
foo :: _ -> _
foo x = x
-- Inferred: forall t. t -> t

filter' :: _
filter' = filter -- has type forall a. (a -> Bool) -> [a] -> [a]
-- Inferred: (a -> Bool) -> [a] -> [a]
```

Named Wildcards

NamedWildCards

Since 7.10.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow naming of wildcards (e.g. _x) in type signatures.

Type wildcards can also be named by giving the underscore an identifier as suffix, i.e. _a. These are called *named wildcards*. All occurrences of the same named wildcard within one type signature will unify to the same type. For example:

```
f :: _x -> _x
f ('c', y) = ('d', error "Urk")
-- Inferred: forall t. (Char, t) -> (Char, t)
```

The named wildcard forces the argument and result types to be the same. Lacking a signature, GHC would have inferred forall a b. (Char, a) -> (Char, b). A named wildcard can be mentioned in constraints, provided it also occurs in the monotype part of the type signature to make sure that it unifies with something:

```
somethingShowable :: Show _x => _x -> _
somethingShowable x = show x
-- Inferred type: Show a => a -> String

somethingShowable' :: Show _x => _x -> _
somethingShowable' x = show (not x)
-- Inferred type: Bool -> String
```

Besides an extra-constraints wildcard (see *Extra-Constraints Wildcard* (page 523)), only named wildcards can occur in the constraints, e.g. the _x in Show _x.

When *ScopedTypeVariables* (page 512) is on, the named wildcards of a function signature scope over the function body just like explicitly-forall'd type variables (*Lexically scoped type variables* (page 512)), even though there is no explicit forall. For example:

```
f :: _a -> _a
f x = let g :: _a -> _a
          g = ...
      in ...
```

Here the named wildcard _a scopes over the body of f, thereby binding the occurrences of _a in the signature of g. All four occurrences stand for the same type.

Named wildcards *should not be confused with type variables*. Even though syntactically similar, named wildcards can unify with monotypes as well as be generalised over (and behave as type variables).

In the first example above, _x is generalised over (and is effectively replaced by a fresh type variable a). In the second example, _x is unified with the Bool type, and as Bool implements the Show type class, the constraint Show Bool can be simplified away.

By default, GHC (as the Haskell 2010 standard prescribes) parses identifiers starting with an underscore in a type as type variables. To treat them as named wildcards, the *NamedWildCards* (page 522) extension should be enabled. The example below demonstrated the effect.

```
foo :: _a -> _a
foo _ = False
```

Compiling this program without enabling *NamedWildCards* (page 522) produces the following error message complaining about the type variable _a no matching the actual type Bool.

```
Test.hs:5:9: error:
    • Couldn't match expected type ‘_a’ with actual type ‘Bool’
      ‘_a’ is a rigid type variable bound by
        the type signature for:
          foo :: forall _a. _a -> _a
        at Test.hs:4:8
    • In the expression: False
      In an equation for ‘foo’: foo _ = False
    • Relevant bindings include foo :: _a -> _a (bound at Test.hs:5:1)
```

Compiling this program with *NamedWildCards* (page 522) (as well as *PartialTypeSignatures* (page 520)) enabled produces the following error message reporting the inferred type of the named wildcard _a.

```
Test.hs:4:8: warning: [-Wpartial-type-signatures]
    • Found type wildcard ‘_a’ standing for ‘Bool’
    • In the type signature:
        foo :: _a -> _a
    • Relevant bindings include
        foo :: Bool -> Bool (bound at Test.hs:5:1)
```

Extra-Constraints Wildcard

The third kind of wildcard is the *extra-constraints wildcard*. The presence of an extra-constraints wildcard indicates that an arbitrary number of extra constraints may be inferred during type checking and will be added to the type signature. In the example below, the extra-constraints wildcard is used to infer three extra constraints.

```
arbitCs :: _ => a -> String
arbitCs x = show (succ x) ++ show (x == x)
-- Inferred:
--   forall a. (Enum a, Eq a, Show a) => a -> String
-- Error:
Test.hs:5:12: error:
    Found constraint wildcard ‘_’ standing for ‘(Show a, Eq a, Enum a)’
    To use the inferred type, enable PartialTypeSignatures
    In the type signature:
      arbitCs :: _ => a -> String
```

An extra-constraints wildcard shouldn't prevent the programmer from already listing the constraints they know or want to annotate, e.g.

```
-- Also a correct partial type signature:
arbitCs' :: (Enum a, _) => a -> String
arbitCs' x = arbitCs x
-- Inferred:
--   forall a. (Enum a, Show a, Eq a) => a -> String
-- Error:
Test.hs:9:22: error:
```

(continues on next page)

(continued from previous page)

```
    Found constraint wildcard '_' standing for '()'
    To use the inferred type, enable PartialTypeSignatures
    In the type signature:
      arbitCs' :: (Enum a, _) => a -> String
```

An extra-constraints wildcard can also lead to zero extra constraints to be inferred, e.g.

```
noCs :: _ => String
noCs = "noCs"
-- Inferred: String
-- Error:
Test.hs:13:9: error:
    Found constraint wildcard '_' standing for '()'
    To use the inferred type, enable PartialTypeSignatures
    In the type signature:
      noCs :: _ => String
```

As a single extra-constraints wildcard is enough to infer any number of constraints, only one is allowed in a type signature and it should come last in the list of constraints.

Extra-constraints wildcards cannot be named.

6.11.6.2 Where can they occur?

Partial type signatures are allowed for bindings, pattern and expression signatures, except that extra-constraints wildcards are not supported in pattern or expression signatures. In the following example a wildcard is used in each of the three possible contexts.

```
{-# LANGUAGE ScopedTypeVariables #-}
foo :: _
foo (x :: _) = (x :: _)
-- Inferred: forall w_. w_ -> w_
```

Anonymous and named wildcards *can* occur on the left hand side of a type or data instance declaration; see *Wildcards on the LHS of data and type family instances* (page 347).

Anonymous wildcards are also allowed in visible type applications/ visible kind applications (*Visible type application* (page 386)). If you want to specify only the second type argument to `wurble`, then you can say `wurble @_ @Int` where the first argument is a wildcard.

Standalone `deriving` declarations permit the use of a single, extra-constraints wildcard, like so:

```
deriving instance _ => Eq (Foo a)
```

This denotes a derived `Eq (Foo a)` instance where the context is inferred, in much the same way that ordinary `deriving` clauses do. Any other use of wildcards in a standalone `deriving` declaration is prohibited.

In all other contexts, type wildcards are disallowed, and a named wildcard is treated as an ordinary type variable. For example:

```
class C _ where ...          -- Illegal
instance Eq (T _)            -- Illegal (currently; would actually make sense)
instance Eq _a => Eq (T _a)  -- Perfectly fine, same as  Eq a => Eq (T a)
```

Partial type signatures can also be used in *Template Haskell* (page 527) splices.

- Declaration splices: partial type signature are fully supported.

```
{-# LANGUAGE TemplateHaskell, NamedWildCards #-}
$( [d| foo :: _ => _a -> _a -> _
       foo x y = x == y|] )
```

- Expression splices: anonymous and named wildcards can be used in expression signatures. Extra-constraints wildcards are not supported, just like in regular expression signatures.

```
{-# LANGUAGE TemplateHaskell, NamedWildCards #-}
$( [e| foo = (Just True :: _m _) |] )
```

- Typed expression splices: the same wildcards as in (untyped) expression splices are supported.
- Pattern splices: anonymous and named wildcards can be used in pattern signatures. Note that *ScopedTypeVariables* (page 512) has to be enabled to allow pattern signatures. Extra-constraints wildcards are not supported, just like in regular pattern signatures.

```
{-# LANGUAGE TemplateHaskell, ScopedTypeVariables #-}
foo $( [p| (x :: _) |] ) = x
```

- Type splices: only anonymous wildcards are supported in type splices. Named and extra-constraints wildcards are not.

```
{-# LANGUAGE TemplateHaskell #-}
foo :: $( [t| _ |] ) -> a
foo x = x
```

6.12 Bindings and generalisation

6.12.1 Switching off the Monomorphism Restriction

NoMonomorphismRestriction

Default on

Since 6.8.1

Prevents the compiler from applying the monomorphism restriction to bindings lacking explicit type signatures.

Haskell's monomorphism restriction (see Section 4.5.5 of the Haskell Report) can be completely switched off by *NoMonomorphismRestriction* (page 525). Since GHC 7.8.1, the monomorphism restriction is switched off by default in GHCi's interactive options (see *Setting options for interactive evaluation only* (page 57)).

6.12.2 Let-generalisation

MonoLocalBinds

Implied by *TypeFamilies* (page 338), *GADTs* (page 335)

Since 6.12.1

Status Included in *GHC2024* (page 270)

Infer less polymorphic types for local bindings by default.

An ML-style language usually generalises the type of any let-bound or where-bound variable, so that it is as polymorphic as possible. With the extension *MonoLocalBinds* (page 526) GHC implements a slightly more conservative policy, for reasons descibed in Section 4.2 of OutsideIn(X): Modular type inference with local assumptions, and a related blog post.

The extension *MonoLocalBinds* (page 526) is implied by *TypeFamilies* (page 338) and *GADTs* (page 335). You can switch it off again with *NoMonoLocalBinds* (page 526) but type inference becomes less predictable if you do so. (Read the paper!)

To a first approximation, with *MonoLocalBinds* (page 526) *top-level bindings are generalised, but local (i.e. nested) bindings are not*. The idea is that, at top level, the type environment has no free type variables, and so the difficulties described in these papers do not arise. But GHC implements a slightly more complicated rule because, for stylistic reasons, programmers sometimes write local bindings that make no use of local variables, so the binding could equally well be top-level. It seems reasonable to generalise these.

So here are the exact rules used by MonoLocalBinds. With MonoLocalBinds, a binding group will be *generalised* if and only if

- It is a top-level binding group, or
- Each of its free variables (excluding the variables bound by the group itself) is *closed* (see next bullet), or
- Any of its binders has a partial type signature (see Partial Type Signatures). Adding a partial type signature `f :: _`, (or, more generally, `f :: _ => _`) provides a per-binding way to ask GHC to perform let-generalisation, even though MonoLocalBinds is on.

Even if the binding is generalised, it may not be generalised over all its free type variables, either because it mentions locally-bound variables, or because of the Monomorphism Restriction (Haskell Report, Section 4.5.5)

Closed variables. The key idea is that: *if a variable is closed, then its type definitely has no free type variables*. A variable `f` is called *closed* if and only if

- The variable `f` is imported from another module, or
- The variable `f` is let-bound, and one of the following holds:
  - `f` has an explicit, complete (i.e. not partial) type signature that has no free type variables, or
  - its binding group is generalised over all its free type variables, so that `f`'s type has no free type variables.

Note that a signature like f :: a -> a is equivalent to `f :: forall a. a -> a`, assuming `a` is not in scope. Hence `f` is closed, since it has a complete type signature with no free variables.

Example 1

```
g v = ...
    where
      f1 x = x+1
      f2 y = f1 (y*2)
```

`f1` has free variable `(+)`, but it is imported and hence closed. So `f1`'s binding is generalised. As a result, its type `f1 :: forall a. Num a => a -> a` has no free type variables, so `f1` is closed. Hence `f2`'s binding is generalised (since its free variables, `f1` and `(*)` are both closed).

Example 2

```
f3 x = let g y = x+y in ....
```

The binding for `g` has a free variable `x` that is lambda-bound, and hence not closed. So `g`'s binding is not generalised.

Top-level bindings. The Monomorphism Restriction can cause even top-level bindings not to be generalised, and hence even the top-level type environment can have free type variables. However, top-level bindings are nevertheless always generalised. To see why, consider

```
module M( f ) where
  x = 5
  f v = (v,x)
```

The binding `x=5` falls under the Monomorphism Restriction, so that binding is not generalised, and hence `f`'s binding is not closed. If, as a result, we did not generalise `f`, we would end up exporting `f :: Any -> (Any, Integer)`, defaulting `x`'s type to *Integer* and `v`'s type to `Any`. This is counter-intuitive and undesirable, so we always generalise top-level bindings.

6.13 Template Haskell

Template Haskell allows you to do compile-time meta-programming in Haskell. The background to the main technical innovations is discussed in "Template Meta-programming for Haskell" (Proc Haskell Workshop 2002).

The Template Haskell page on the GHC Wiki has a wealth of information. You may also consult the Haddock reference documentation Language.Haskell.TH. Many changes to the original design are described in Notes on Template Haskell version 2. Not all of these changes are in GHC, however.

The first example from that paper is set out below (*A Template Haskell Worked Example* (page 535)) as a worked example to help get you started.

The documentation here describes the realisation of Template Haskell in GHC. It is not detailed enough to understand Template Haskell; see the Wiki page.

6.13.1 Syntax

TemplateHaskell

Implies *TemplateHaskellQuotes* (page 528)

Since 6.0. Typed splices introduced in GHC 7.8.1.

Enable Template Haskell's splice and quotation syntax.

TemplateHaskellQuotes

Since 8.0.1

Enable only Template Haskell's quotation syntax.

Template Haskell has the following new syntactic constructions. You need to use the extension *TemplateHaskell* (page 528) to switch these syntactic extensions on. Alternatively, the *TemplateHaskellQuotes* (page 528) extension can be used to enable the quotation subset of Template Haskell (i.e. without top-level splices). The *TemplateHaskellQuotes* (page 528) extension is considered safe under *Safe Haskell* (page 577) while *TemplateHaskell* (page 528) is not.

- A splice is written `$x`, where `x` is an arbitrary expression. There must be no space between the "$" and the expression. This use of "$" overrides its meaning as an infix operator, just as "M.x" overrides the meaning of "." as an infix operator. If you want the infix operator, put spaces around it.

  A top-level splice can occur in place of

  - an expression; the spliced expression must have type `Q Exp`
  - a pattern; the spliced pattern must have type `Q Pat`
  - a type; the spliced expression must have type `Q Type`
  - a list of declarations at top level; the spliced expression must have type `Q [Dec]`

  Inside a splice you can only call functions defined in imported modules, not functions defined elsewhere in the same module. Note that declaration splices are not allowed anywhere except at top level (outside any other declarations).

  The `Q` monad is a monad defined in Language.Haskell.TH.Syntax which supports several useful operations during code generation such as reporting errors or looking up identifiers in the environment.

- A expression quotation is written in Oxford brackets, thus:

  - `[| ... |]`, or `[e| ... |]`, where the "..." is an expression; the quotation has type `Quote m => m Exp`.
  - `[d| ... |]`, where the "..." is a list of top-level declarations; the quotation has type `Quote m => m [Dec]`.
  - `[t| ... |]`, where the "..." is a type; the quotation has type `Quote m => m Type`.
  - `[p| ... |]`, where the "..." is a pattern; the quotation has type `Quote m => m Pat`.

  The `Quote` type class (Language.Haskell.TH.Syntax.Quote) is the minimal interface necessary to implement the desugaring of quotations. The `Q` monad is an instance of `Quote` but contains many more operations which are not needed for defining quotations.

  See *Where can they occur?* (page 524) for using partial type signatures in quotations.

- Splices can be nested inside quotation brackets. For example the fragment representing `1 + 2` can be constructed using nested splices:

```
oneC, twoC, plusC  :: Quote m => m Exp
oneC = [| 1 |]

twoC = [| 2 |]

plusC = [| $oneC + $twoC |]
```

- The precise type of a quotation depends on the types of the nested splices inside it:

```
-- Add a redundant constraint to demonstrate that constraints on the
-- monad used to build the representation are propagated when using nested
-- splices.
f :: (Quote m, C m) => m Exp
f = [| 5 | ]

-- f is used in a nested splice so the constraint on f, namely C, is propagated
-- to a constraint on the whole representation.
g :: (Quote m, C m) => m Exp
g = [| $f + $f |]
```

  Remember, a top-level splice still requires its argument to be of type `Q Exp`. So then splicing in `g` will cause `m` to be instantiated to `Q`:

```
h :: Int
h = $(g) -- m ~ Q
```

- A *typed* expression splice is written `$$x`, where `x` is is an arbitrary expression.

  A top-level typed expression splice can occur in place of an expression; the spliced expression must have type `Code Q a`

  NOTE: Currently typed splices may inhibit the unused identifier warning for identifiers in scope. See #16524.

- A *typed* expression quotation is written as `[|| ... ||]`, or `[e|| ... ||]`, where the "…" is an expression; if the "…" expression has type `a`, then the quotation has type `Quote m => Code m a`.

  It is possible to extract a value of type `m Exp` from `Code m a` using the `unTypeCode :: Code m a -> m Exp` function.

- A quasi-quotation can appear in a pattern, type, expression, or declaration context and is also written in Oxford brackets:

  - `[varid| ... |]`, where the "…" is an arbitrary string; a full description of the quasi-quotation facility is given in *Template Haskell Quasi-quotation* (page 538).

- A name can be quoted with either one or two prefix single quotes:

  - `'f` has type `Name`, and names the function `f`. Similarly `'C` has type `Name` and names the data constructor `C`. In general `'`⟨thing⟩ interprets ⟨thing⟩ in an expression context.

    A name whose second character is a single quote cannot be quoted in exactly this way, because it will be parsed instead as a quoted character. For example, if the function is called `f'7` (which is a legal Haskell identifier), an attempt to quote it as `'f'7` would be parsed as the character literal `'f'` followed by the numeric literal `7`. As for promoted constructors (*Distinguishing between types and constructors* (page 359)), the workaround is to add a space between the quote and the name. The name of the function `f'7` is thus written `' f'7`.

- `''T` has type `Name`, and names the type constructor `T`. That is, `''`⟨thing⟩ interprets ⟨thing⟩ in a type context.

These `Names` can be used to construct Template Haskell expressions, patterns, declarations etc. They may also be given as an argument to the `reify` function.

- It is possible for a splice to expand to an expression that contain names which are not in scope at the site of the splice. As an example, consider the following code:

```
module Bar where

import Language.Haskell.TH

add1 :: Quote m => Int -> m Exp
add1 x = [| x + 1 |]
```

Now consider a splice using `add1` in a separate module:

```
module Foo where

import Bar

two :: Int
two = $(add1 1)
```

Template Haskell cannot know what the argument to `add1` will be at the function's definition site, so a lifting mechanism is used to promote `x` into a value of type `Quote m => m Exp`. This functionality is exposed to the user as the `Lift` typeclass in the `Language.Haskell.TH.Syntax` module. If a type has a `Lift` instance, then any of its values can be lifted to a Template Haskell expression:

```
class Lift t where
    lift :: Quote m => t -> m Exp
    liftTyped :: Quote m => t -> Code m t
```

In general, if GHC sees an expression within Oxford brackets (e.g., `[| foo bar |]`, then GHC looks up each name within the brackets. If a name is global (e.g., suppose `foo` comes from an import or a top-level declaration), then the fully qualified name is used directly in the quotation. If the name is local (e.g., suppose `bar` is bound locally in the function definition `mkFoo bar = [| foo bar |]`), then GHC uses `lift` on it (so GHC pretends `[| foo bar |]` actually contains `[| foo $(lift bar) |]`). Local names, which are not in scope at splice locations, are actually evaluated when the quotation is processed.

The `template-haskell` library provides `Lift` instances for many common data types. Furthermore, it is possible to derive `Lift` instances automatically by using the *`DeriveLift`* (page 446) language extension. See *Deriving Lift instances* (page 446) for more information.

- You may omit the `$(...)` in a top-level declaration splice. Simply writing an expression (rather than a declaration) implies a splice. For example, you can write

```
module Foo where
import Bar

```

(continues on next page)

(continued from previous page)

```
f x = x

$(deriveStuff 'f)   -- Uses the $(...) notation

g y = y+1

deriveStuff 'g      -- Omits the $(...)

h z = z-1
```

This abbreviation makes top-level declaration slices quieter and less intimidating.

- Pattern splices introduce variable binders but scoping of variables in expressions inside the pattern's scope is only checked when a splice is run. Note that pattern splices that occur outside of any quotation brackets are run at compile time. Pattern splices occurring inside a quotation bracket are *not* run at compile time; they are run when the bracket is spliced in, sometime later. For example,

```
mkPat :: Quote m => m Pat
mkPat = [p| (x, y) |]

-- in another module:
foo :: (Char, String) -> String
foo $(mkPat) = x : z

bar :: Quote m => m Exp
bar = [| \ $(mkPat) -> x : w |]
```

  will fail with `z` being out of scope in the definition of `foo` but it will *not* fail with `w` being out of scope in the definition of `bar`. That will only happen when `bar` is spliced.

- A pattern quasiquoter *may* generate binders that scope over the right-hand side of a definition because these binders are in scope lexically. For example, given a quasiquoter `haskell` that parses Haskell, in the following code, the `y` in the right-hand side of `f` refers to the `y` bound by the `haskell` pattern quasiquoter, *not* the top-level `y = 7`.

```
y :: Int
y = 7

f :: Int -> Int -> Int
f n = \ [haskell|y|] -> y+n
```

- Top-level declaration splices break up a source file into *declaration groups*. A *declaration group* is the group of declarations created by a top-level declaration splice, plus those following it, down to but not including the next top-level declaration splice. N.B. only top-level splices delimit declaration groups, not expression splices. The first declaration group in a module includes all top-level definitions down to but not including the first top-level declaration splice.

  Each group is compiled just like a separately compiled module. That is:

  - Later groups can "see" declarations, and instance declarations, from earlier groups;
  - But earlier groups cannot "see" declarations, or instance declarations, from later groups.

  Each declaration group is mutually recursive only within the group. Declaration groups can refer to definitions within previous groups, but not later ones.

Accordingly, the type environment seen by `reify` includes all the top-level declarations up to the end of the immediately preceding declaration group, but no more.

Unlike normal declaration splices, declaration quasiquoters do not cause a break. These quasiquoters are expanded before the rest of the declaration group is processed, and the declarations they generate are merged into the surrounding declaration group. Consequently, the type environment seen by `reify` from a declaration quasiquoter will not include anything from the quasiquoter's declaration group.

Concretely, consider the following code

```
module M where

import ...

f x = x

$(th1 4)

h y = k y y $(blah1)

[qq|blah|]

k x y z = x + y + z

$(th2 10)

w z = $(blah2)
```

In this example, a `reify` inside…

1. The splice `$(th1 ...)` would see the definition of `f` - the splice is top-level and thus all definitions in the previous declaration group are visible (that is, all definitions in the module up-to, but not including, the splice itself).
2. The splice `$(blah1)` cannot refer to the function `w` - `w` is part of a later declaration group, and thus invisible, similarly, `$(blah1)` cannot see the definition of `h` (since it is part of the same declaration group as `$(blah1)`. However, the splice `$(blah1)` can see the definition of `f` (since it is in the immediately preceding declaration group).
3. The splice `$(th2 ...)` would see the definition of `f`, all the bindings created by `$(th1 ...)`, the definition of `h` and all bindings created by `[qq|blah|]` (they are all in previous declaration groups).
4. The body of `h` *can* refer to the function `k` appearing on the other side of the declaration quasiquoter, as quasiquoters do not cause a declaration group to be broken up.
5. The `qq` quasiquoter would be able to see the definition of `f` from the preceding declaration group, but not the definitions of `h` or `k`, or any definitions from subsequent declaration groups.
6. The splice `$(blah2)` would see the same definitions as the splice `$(th2 ...)` (but *not* any bindings it creates).

Note that since an expression splice is unable to refer to declarations in the same declaration group, we can introduce a top-level (empty) splice to break up the declaration group

```haskell
module M where

data D = C1 | C2

f1 = $(th1 ...)

$(return [])

f2 = $(th2 ...)
```

Here

1. The splice `$(th1 ...)` *cannot* refer to `D` - it is in the same declaration group.
2. The declaration group containing `D` is terminated by the empty top-level declaration splice `$(return [])` (recall, `Q` is a Monad, so we may simply `return` the empty list of declarations).
3. Since the declaration group containing `D` is in the previous declaration group, the splice `$(th2 ...)` *can* refer to `D`.

Note that in some cases, the presence or absence of top-level declaration splices can affect the *runtime* behavior of the surrounding code, because the resolution of instances may differ depending on their visiblity. One case where this arises is with *incoherent instances* (page 486)

```haskell
module Main where

main :: IO ()
main = do
  let i :: Int
      i = 42
  putStrLn (m1 i)
  putStrLn (m2 i)

class C1 a where
  m1 :: a -> String

instance {-# INCOHERENT #-} C1 a where
  m1 _ = "C1 incoherent"

instance C1 Int where
  m1 = show

class C2 a where
  m2 :: a -> String

instance {-# INCOHERENT #-} C2 a where
  m2 _ = "C2 incoherent"

$(return [])

instance C2 Int where
  m2 = show
```

Here, `C1` and `C2` are the same classes with nearly identical instances. The only significant differences between `C1` and `C2`, aside from the minor name change, is that all of `C1`'s instances are defined within the same declaration group, whereas the `C2 Int` instance is put in a separate declaration group from the incoherent `C2 a` instance. This has an impact on the runtime behavior of the `main` function

```
$ runghc Main.hs
42
C2 incoherent
```

Note that `m1 i` returns `"42"`, but `m2 i` returns `"C2 incoherent"`. When each of these expressions are typechecked, GHC must figure out which `C1 Int` and `C2 Int` instances to use:

1. When resolving the `C1 Int` instance, GHC discovers two possible instances in the same declaration group: the incoherent `C1 a` instance and the non-incoherent `C1 Int` instance. According to the instance search rules described in *Overlapping instances* (page 486), because there is exactly one non-incoherent instance to pick, GHC will choose the `C1 Int` instance. As a result, `m1 i` will be equivalent to `show i` (i.e., `"42"`).
2. When resolving the `C2 Int` instance, GHC only discovers one instance in the same declaration group: the incoherent `C2 a` instance. Note that GHC does *not* see the `C2 Int` instance, as that is in a later declaration group that is made separate by the intervening declaration splice. As a result, GHC will choose the `C2 a` instance, making `m2 i` equivalent to `"C2 incoherent"`.

- Expression quotations accept most Haskell language constructs. However, there are some GHC-specific extensions which expression quotations currently do not support, including
  - Type holes in typed splices (see #10945 and #10946)

(Compared to the original paper, there are many differences of detail. The syntax for a declaration splice uses "`$`" not "`splice`". The type of the enclosed expression must be `Quote m => m [Dec]`, not `[Q Dec]`. Typed expression splices and quotations are supported.)

`-fenable-th-splice-warnings`
Template Haskell splices won't be checked for warnings, because the code causing the warning might originate from a third-party library and possibly was not written by the user. If you want to have warnings for splices anyway, pass *`-fenable-th-splice-warnings`* (page 534).

6.13.2 Using Template Haskell

- The data types and monadic constructor functions for Template Haskell are in the library Language.Haskell.TH.Syntax.
- You can only run a function at compile time if it is imported from another module. That is, you can't define a function in a module, and call it from within a splice in the same module. (It would make sense to do so, but it's hard to implement.)
- You can only run a function at compile time if it is imported from another module *that is not part of a mutually-recursive group of modules that includes the module currently being compiled*. Furthermore, all of the modules of the mutually-recursive group must be reachable by non-SOURCE imports from the module where the splice is to be run.

  For example, when compiling module A, you can only run Template Haskell functions imported from B if B does not import A (directly or indirectly). The reason should be clear: to run B we must compile and run A, but we are currently type-checking A.

- If you are building GHC from source, you need at least a stage-2 bootstrap compiler to run Template Haskell splices and quasi-quotes. A stage-1 compiler will only accept regular quotes of Haskell. Reason: TH splices and quasi-quotes compile and run a program, and then looks at the result. So it's important that the program it compiles produces results whose representations are identical to those of the compiler itself.

Template Haskell works in any mode (*--make* (page 68), *--interactive* (page 68), or file-at-a-time). There used to be a restriction to the former two, but that restriction has been lifted.

6.13.3 Viewing Template Haskell generated code

The flag *-ddump-splices* (page 257) shows the expansion of all top-level declaration splices, both typed and untyped, as they happen. As with all dump flags, the default is for this output to be sent to stdout. For a non-trivial program, you may be interested in combining this with the *-ddump-to-file* (page 255) flag (see *Dumping out compiler intermediate structures* (page 255). For each file using Template Haskell, this will show the output in a `.dump-splices` file.

The flag *-dth-dec-file* (page 257) dumps the expansions of all top-level TH declaration splices, both typed and untyped, in the file `M.th.hs` for each module *M* being compiled. Note that other types of splices (expressions, types, and patterns) are not shown. Application developers can check this into their repository so that they can grep for identifiers that were defined in Template Haskell. This is similar to using *-ddump-to-file* (page 255) with *-ddump-splices* (page 257) but it always generates a file instead of being coupled to *-ddump-to-file* (page 255). The format is also different: it does not show code from the original file, instead it only shows generated code and has a comment for the splice location of the original file.

Below is a sample output of *-ddump-splices* (page 257)

```
TH_pragma.hs:(6,4)-(8,26): Splicing declarations
  [d| foo :: Int -> Int
      foo x = x + 1 |]
======>
  foo :: Int -> Int
  foo x = (x + 1)
```

Below is the output of the same sample using *-dth-dec-file* (page 257)

```
-- TH_pragma.hs:(6,4)-(8,26): Splicing declarations
foo :: Int -> Int
foo x = (x + 1)
```

6.13.4 A Template Haskell Worked Example

To help you get over the confidence barrier, try out this skeletal worked example. First cut and paste the two modules below into `Main.hs` and `Printf.hs`:

```
{- Main.hs -}
module Main where

-- Import our template "pr"
import Printf ( pr )
```

(continues on next page)

(continued from previous page)

```
-- The splice operator $ takes the Haskell source code
-- generated at compile time by "pr" and splices it into
-- the argument of "putStrLn".
main = putStrLn ( $(pr "Hello") )


{- Printf.hs -}
module Printf where

-- Skeletal printf from the paper.
-- It needs to be in a separate module to the one where
-- you intend to use it.

-- Import some Template Haskell syntax
import Language.Haskell.TH

-- Describe a format string
data Format = D | S | L String

-- Parse a format string.  This is left largely to you
-- as we are here interested in building our first ever
-- Template Haskell program and not in building printf.
parse :: String -> [Format]
parse s   = [ L s ]

-- Generate Haskell source code from a parsed representation
-- of the format string.  This code will be spliced into
-- the module which calls "pr", at compile time.
gen :: Quote m => [Format] -> m Exp
gen [D]   = [| \n -> show n |]
gen [S]   = [| \s -> s |]
gen [L s] = stringE s

-- Here we generate the Haskell code for the splice
-- from an input format string.
pr :: Quote m => String -> m Exp
pr s = gen (parse s)
```

Now run the compiler,

```
$ ghc --make -XTemplateHaskell main.hs -o main
```

Run main and here is your output:

```
$ ./main
Hello
```

6.13.5 Template Haskell quotes and Rebindable Syntax

Rebindable syntax does not play well with untyped TH quotes: applying the rebindable syntax rules would go against the lax nature of untyped quotes that are accepted even in the presence of unbound identifiers (see #18102). Applying the rebindable syntax rules to them would force the code that defines the said quotes to have all the necessary functions (e.g `ifThenElse` or `fromInteger`) in scope, instead of delaying the resolution of those symbols to the code that splices the quoted Haskell syntax, as is usually done with untyped TH. For this reason, even if a module has untyped TH quotes with `RebindableSyntax` enabled, GHC turns off rebindable syntax while processing the quotes. The code that splices the quotes is however free to turn on `RebindableSyntax` to have the usual rules applied to the resulting code.

Typed TH quotes on the other hand are perfectly compatible with the eager application of rebindable syntax rules, and GHC will therefore process any such quotes according to the rebindable syntax rules whenever the `RebindableSyntax` extension is turned on in the modules where such quotes appear.

6.13.6 Using Template Haskell with Profiling

Template Haskell relies on GHC's built-in bytecode compiler and interpreter to run the splice expressions. The bytecode interpreter runs the compiled expression on top of the same runtime on which GHC itself is running; this means that the compiled code referred to by the interpreted expression must be compatible with this runtime, and in particular this means that object code that is compiled for profiling *cannot* be loaded and used by a splice expression, because profiled object code is only compatible with the profiling version of the runtime.

This causes difficulties if you have a multi-module program containing Template Haskell code and you need to compile it for profiling, because GHC cannot load the profiled object code and use it when executing the splices.

Fortunately GHC provides two workarounds.

The first option is to compile the program twice:

1. Compile the program or library first the normal way, without *`-prof`* (page 656).
2. Then compile it again with *`-prof`* (page 656), and additionally use `-osuf p_o` to name the object files differently (you can choose any suffix that isn't the normal object suffix here). GHC will automatically load the object files built in the first step when executing splice expressions. If you omit the *`-osuf ⟨suffix⟩`* (page 203) flag when building with *`-prof`* (page 656) and Template Haskell is used, GHC will emit an error message.

The second option is to add the flag *`-fexternal-interpreter`* (page 60) (see *Running the interpreter in a separate process* (page 60)), which runs the interpreter in a separate process, wherein it can load and run the profiled code directly. There's no need to compile the code twice, just add *`-fexternal-interpreter`* (page 60) and it should just work. (this option is experimental in GHC 8.0.x, but it may become the default in future releases).

6.13.7 Template Haskell Quasi-quotation

QuasiQuotes

Since 6.10.1

Enable Template Haskell Quasi-quotation syntax.

Quasi-quotation allows patterns and expressions to be written using programmer-defined concrete syntax; the motivation behind the extension and several examples are documented in "Why It's Nice to be Quoted: Quasiquoting for Haskell" (Proc Haskell Workshop 2007). The example below shows how to write a quasiquoter for a simple expression language.

Here are the salient features

- A quasi-quote has the form `[quoter| string |]`.
  - The ⟨quoter⟩ must be the name of an imported quoter, either qualified or unqualified; it cannot be an arbitrary expression.
  - The ⟨quoter⟩ cannot be "`e`", "`t`", "`d`", or "`p`", since those overlap with Template Haskell quotations.
  - There must be no spaces in the token `[quoter|`.
  - The quoted ⟨string⟩ can be arbitrary, and may contain newlines.
  - The quoted ⟨string⟩ finishes at the first occurrence of the two-character sequence `"|]"`. Absolutely no escaping is performed. If you want to embed that character sequence in the string, you must invent your own escape convention (such as, say, using the string `"|~]"` instead), and make your quoter function interpret `"|~]"` as `"|]"`. One way to implement this is to compose your quoter with a pre-processing pass to perform your escape conversion. See the discussion in #5348 for details.
- A quasiquote may appear in place of
  - An expression
  - A pattern
  - A type
  - A top-level declaration

  (Only the first two are described in the paper.)
- A quoter is a value of type Language.Haskell.TH.Quote.QuasiQuoter, which is defined thus:

```
data QuasiQuoter = QuasiQuoter { quoteExp  :: String -> Q Exp,
                                 quotePat  :: String -> Q Pat,
                                 quoteType :: String -> Q Type,
                                 quoteDec  :: String -> Q [Dec] }
```

  That is, a quoter is a tuple of four parsers, one for each of the contexts in which a quasi-quote can occur.
- A quasi-quote is expanded by applying the appropriate parser to the string enclosed by the Oxford brackets. The context of the quasi-quote (expression, pattern, type, declaration) determines which of the parsers is called.
- Unlike normal declaration splices of the form `$(...)`, declaration quasi-quotes do not cause a declaration group break. See *Syntax* (page 528) for more information.

Warning: `QuasiQuotes` (page 538) introduces an unfortunate ambiguity with list comprehension syntax. Consider the following,

```
let x = [v| v <- [0..10]]
```

Without `QuasiQuotes` (page 538) this is parsed as a list comprehension. With `QuasiQuotes` (page 538) this is parsed as a quasi-quote; however, this parse will fail due to the lack of a closing `|]`. See #11679.

The example below shows quasi-quotation in action. The quoter `expr` is bound to a value of type `QuasiQuoter` defined in module `Expr`. The example makes use of an antiquoted variable `n`, indicated by the syntax `'int:n` (this syntax for anti-quotation was defined by the parser's author, *not* by GHC). This binds `n` to the integer value argument of the constructor `IntExpr` when pattern matching. Please see the referenced paper for further details regarding anti-quotation as well as the description of a technique that uses SYB to leverage a single parser of type `String -> a` to generate both an expression parser that returns a value of type `Q Exp` and a pattern parser that returns a value of type `Q Pat`.

Quasiquoters must obey the same stage restrictions as Template Haskell, e.g., in the example, `expr` cannot be defined in `Main.hs` where it is used, but must be imported.

```
{- ------------- file Main.hs --------------- -}
module Main where

import Expr

main :: IO ()
main = do { print $ eval [expr|1 + 2|]
          ; case IntExpr 1 of
              { [expr|'int:n|] -> print n
              ;  _              -> return ()
              }
          }


{- ------------- file Expr.hs --------------- -}
module Expr where

import qualified Language.Haskell.TH as TH
import Language.Haskell.TH.Quote

data Expr  =  IntExpr Integer
           |  AntiIntExpr String
           |  BinopExpr BinOp Expr Expr
           |  AntiExpr String
    deriving(Show, Typeable, Data)

data BinOp  =  AddOp
            |  SubOp
            |  MulOp
            |  DivOp
    deriving(Show, Typeable, Data)

eval :: Expr -> Integer
eval (IntExpr n)        = n
eval (BinopExpr op x y) = (opToFun op) (eval x) (eval y)
```

(continues on next page)

(continued from previous page)

```
  where
    opToFun AddOp = (+)
    opToFun SubOp = (-)
    opToFun MulOp = (*)
    opToFun DivOp = div

expr = QuasiQuoter { quoteExp = parseExprExp, quotePat =  parseExprPat }

-- Parse an Expr, returning its representation as
-- either a Q Exp or a Q Pat. See the referenced paper
-- for how to use SYB to do this by writing a single
-- parser of type String -> Expr instead of two
-- separate parsers.

parseExprExp :: String -> Q Exp
parseExprExp ...

parseExprPat :: String -> Q Pat
parseExprPat ...
```

Now run the compiler:

```
$ ghc --make -XQuasiQuotes Main.hs -o main
```

Run "main" and here is your output:

```
$ ./main
3
1
```

6.14 Bang patterns and Strict Haskell

In high-performance Haskell code (e.g. numeric code) eliminating thunks from an inner loop can be a huge win. GHC supports three extensions to allow the programmer to specify use of strict (call-by-value) evaluation rather than lazy (call-by-need) evaluation.

- Bang patterns (*BangPatterns* (page 541)) makes pattern matching and let bindings stricter.
- Strict data types (*StrictData* (page 542)) makes constructor fields strict by default, on a per-module basis.
- Strict pattern (*Strict* (page 543)) makes all patterns and let bindings strict by default, on a per-module basis.

The latter two extensions are simply a way to avoid littering high-performance code with bang patterns, making it harder to read.

Bang patterns and strict matching do not affect the type system in any way.

6.14.1 Bang patterns

BangPatterns

Since 6.8.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow use of bang pattern syntax.

GHC supports an extension of pattern matching called *bang patterns*, written `!pat`. Bang patterns are available by default as a part of *GHC2021* (page 271).

The main idea is to add a single new production to the syntax of patterns:

```
pat ::= !pat
```

Matching an expression `e` against a pattern `!p` is done by first evaluating `e` (to WHNF) and then matching the result against `p`. Example:

```
f1 !x = True
```

This definition makes `f1` is strict in `x`, whereas without the bang it would be lazy.

Note the following points:

- Bang patterns can be nested:

  ```
  f2 (!x, y) = [x,y]
  ```

  Here, `f2` is strict in `x` but not in `y`.

- Bang patterns can be used in `case` expressions too:

  ```
  g1 x = let y = f x in body
  g2 x = case f x of { y -> body }
  g3 x = case f x of { !y -> body }
  ```

  The functions `g1` and `g2` mean exactly the same thing. But `g3` evaluates `(f x)`, binds `y` to the result, and then evaluates `body`.

- Bang patterns do not have any effect with constructor patterns:

  ```
  f3 !(x,y) = [x,y]
  f4 (x,y)  = [x,y]
  ```

  Here, `f3` and `f4` are identical; putting a bang before a pattern that forces evaluation anyway does nothing. However, see the caveat below.

- There is one problem with syntactic ambiguity. Consider:

  ```
  f !x = 3
  ```

  Is this a definition of the infix function "`(!)`", or of the "`f`" with a bang pattern? GHC resolves this ambiguity by looking at the surrounding whitespace:

  ```
  a ! b = ...   -- infix operator
  a !b = ...    -- bang pattern
  ```

  See GHC Proposal #229 for the precise rules.

6.14.1.1 Strict bindings

The `BangPatterns` extension furthermore enables syntax for strict `let` or `where` bindings with `!pat = expr`. For example,

```
let !x = e in body
let !(p,q) = e in body
```

In both cases `e` is evaluated before starting to evaluate `body`.

Note the following points:

- A strict binding (with a top level `!`) should not be thought of as a regular pattern binding that happens to have a bang pattern (*Bang patterns* (page 541)) on the LHS. Rather, the top level `!` should be considered part of the let-binding, rather than part of the pattern. This makes a difference when we come to the rules in *Dynamic semantics of bang patterns* (page 545).
- Only a top-level bang (perhaps under parentheses) makes the binding strict; otherwise, it is considered a normal bang pattern. For example,

  ```
  let (!x,[y]) = e in b
  ```

  is equivalent to this:

  ```
  let { t = case e of (x,[y]) -> x `seq` (x,y)
        x = fst t
        y = snd t }
  in b
  ```

  The binding is lazy, but when either `x` or `y` is evaluated by `b` the entire pattern is matched, including forcing the evaluation of `x`.
- Because the `!` in a strict binding is not a bang pattern, it must be visible without looking through pattern synonyms

  ```
  pattern Bang x <- !x
  f1 = let Bang x = y in ...
  f2 = let !x     = y in ...  -- not equivalent to f1
  ```

- Strict bindings are not allowed at the top level of a module.
- See *Semantics of let bindings with bang patterns* (page 545) for the detailed semantics, and the Haskell prime feature description for more discussion and examples.

6.14.2 Strict-by-default data types

StrictData

Since 8.0.1

Make fields of data types defined in the current module strict by default.

Informally the `StrictData` language extension switches data type declarations to be strict by default allowing fields to be lazy by adding a ~ in front of the field.

When the user writes

```
data T = C a
data T' = C' ~a
```

we interpret it as if they had written

```
data T = C !a
data T' = C' a
```

The extension only affects definitions in this module.

The ~ annotation must be written in prefix form:

```
data T = MkT ~Int   -- valid
data T = MkT ~ Int  -- invalid
```

See GHC Proposal #229 for the precise rules.

6.14.3 Strict-by-default pattern bindings

Strict

> **Implies** *StrictData* (page 542)
>
> **Since** 8.0.1

Make bindings in the current module strict by default.

Informally the `Strict` language extension switches functions, data types, and bindings to be strict by default, allowing optional laziness by adding ~ in front of a variable. This essentially reverses the present situation where laziness is default and strictness can be optionally had by adding `!` in front of a variable.

`Strict` implies *StrictData* (page 542).

- **Function definitions**

  When the user writes

  ```
  f x = ...
  ```

  we interpret it as if they had written

  ```
  f !x = ...
  ```

  Adding ~ in front of `x` gives the regular lazy behavior.

  Turning patterns into irrefutable ones requires `~(~p)` when `Strict` is enabled.

- **Let/where bindings**

  When the user writes

  ```
  let x = ...
  let pat = ...
  ```

  we interpret it as if they had written

  ```
  let !x = ...
  let !pat = ...
  ```

Adding ~ in front of x gives the regular lazy behavior. The general rule is that we add an implicit bang on the outermost pattern, unless disabled with ~.

- **Pattern matching in case expressions, lambdas, do-notation, etc**

  The outermost pattern of all pattern matches gets an implicit bang, unless disabled with ~. This applies to case expressions, patterns in lambda, do-notation, list comprehension, and so on. For example

```
case x of (a,b) -> rhs
```

  is interpreted as

```
case x of !(a,b) -> rhs
```

  Since the semantics of pattern matching in case expressions is strict, this usually has no effect whatsoever. But it does make a difference in the degenerate case of variables and newtypes. So

```
case x of y -> rhs
```

  is lazy in Haskell, but with `Strict` is interpreted as

```
case x of !y -> rhs
```

  which evaluates `x`. Similarly, if `newtype Age = MkAge Int`, then

```
case x of MkAge i -> rhs
```

  is lazy in Haskell; but with `Strict` the added bang makes it strict.

  Similarly

```
\ x -> body
do { x <- rhs; blah }
[ e | x <- rhs; blah }
```

  all get implicit bangs on the `x` pattern.

- **Nested patterns**

  Notice that we do *not* put bangs on nested patterns. For example

```
let (p,q) = if flob then (undefined, undefined) else (True, False)
in ...
```

  will behave like

```
let !(p,q) = if flob then (undefined, undefined) else (True,False)
in ...
```

  which will strictly evaluate the right hand side, and bind `p` and `q` to the components of the pair. But the pair itself is lazy (unless we also compile the `Prelude` with `Strict`; see *Modularity* (page 545) below). So `p` and `q` may end up bound to undefined. See also *Dynamic semantics of bang patterns* (page 545) below.

- **Top level bindings**

  are unaffected by `Strict`. For example:

```
x = factorial 20
(y,z) = if x > 10 then True else False
```

Here x and the pattern binding (y,z) remain lazy. Reason: there is no good moment to force them, until first use.

- **Newtypes**

  There is no effect on newtypes, which simply rename existing types. For example:

```
newtype T = C a
f (C x)  = rhs1
g !(C x) = rhs2
```

  In ordinary Haskell, f is lazy in its argument and hence in x; and g is strict in its argument and hence also strict in x. With Strict, both become strict because f's argument gets an implicit bang.

6.14.4 Modularity

Strict and StrictData only affects definitions in the module they are used in. Functions and data types imported from other modules are unaffected. For example, we won't evaluate the argument to Just before applying the constructor. Similarly we won't evaluate the first argument to Data.Map.findWithDefault before applying the function.

This is crucial to preserve correctness. Entities defined in other modules might rely on laziness for correctness (whether functional or performance).

Tuples, lists, Maybe, and all the other types from Prelude continue to have their existing, lazy, semantics.

6.14.5 Dynamic semantics of bang patterns

The semantics of Haskell pattern matching is described in Section 3.17.2 of the Haskell Report. To this description add one extra item 9, saying:

- Matching the pattern !pat against a value v behaves as follows:
  - if v is bottom, the match diverges
  - otherwise, pat is matched against v

Similarly, in Figure 4 of Section 3.17.3, add a new case (w):

```
case v of { !pat -> e; _ -> e' }
   = v `seq` case v of { pat -> e; _ -> e' }
```

That leaves let expressions, whose translation is given in Section 3.12 of the Haskell Report. Replace the "Translation" there with the following one. Given let { bind1 ... bindn } in body:

SPLIT-LAZY

Given a lazy pattern binding p = e, where p is not a variable, and x1...xn are the variables bound by p, and all these binders have lifted type, replace the binding with this (where v is fresh):

```
v = case e of { p -> (x1, ..., xn) }
x1 = case v of { (x1, ..., xn) -> x1 }
...
xn = case v of { (x1, ..., xn) -> xn }``
```

If n=1 (i.e. exactly one variable is bound), the desugaring uses the `Solo` type to make a 1-tuple.

SPLIT-STRICT

Given a strict pattern binding `!p = e`, where `x1...xn` are the variables bound by `p`, and all these binders have lifted type:

1. Replace the binding with this (where `v` is fresh):

   ```
   v = case e of { !p -> (x1, ..., xn) }
   (x1, ..., xn) = v
   ```

2. Replace `body` with ``v `seq` body``.

As in SPLIT-LAZY, if n=1 the desugaring uses the `Solo` type to make a 1-tuple.

This transformation is illegal at the top level of a module (since there is no `body`), so strict bindings are illegal at top level.

The transformation is correct when `p` is a variable `x`, but can be optimised to:

```
let !x = e in body  ==>  let x = e in x `seq` body
```

CASE

Given a non-recursive strict pattern binding `!p = e`, where `x1...xn` are the variables bound by `p`, and any of the binders has unlifted type: replace the binding with nothing at all, and replace `body` with `case e of p -> body`.

This transformation is illegal at the top level of a module, so such bindings are rejected.

The result of this transformation is ill-scoped if any of the binders `x1...xn` appears in `e`; hence the restriction to non-recursive pattern bindings.

Exactly the same transformation applies to a non-recursive lazy pattern binding (i.e. one lacking a top-level `!`) that binds any unlifted variables; but such a binding emits a warning *-Wunbanged-strict-patterns* (page 106). The warning encourages the programmer to make visible the fact that this binding is necessarily strict.

The result will be a (possibly) recursive set of bindings, binding only simple variables on the left hand side. (One could go one step further, as in the Haskell Report and make the recursive bindings non-recursive using `fix`, but we do not do so in Core, and it only obfuscates matters, so we do not do so here.)

The translation is carefully crafted to make bang patterns meaningful for recursive and polymorphic bindings as well as straightforward non-recursive bindings.

Here are some examples of how this translation works. The first expression of each sequence is Haskell source; the subsequent ones are Core.

Here is a simple non-recursive case:

```
let x :: Int     -- Non-recursive
    !x = factorial y
in body

===> (SPLIT-STRICT)
     let x = factorial y in x `seq` body

===> (inline seq)
     let x = factorial y in case x of !x -> body

===> (inline x)
     case factorial y of !x -> body
```

Same again, only with a pattern binding:

```
let !(Just x) = e in body

===> (SPLIT-STRICT)
     let v = case e of !(Just x) -> Solo x
         Solo x = v
     in v `seq` body

===> (SPLIT-LAZY, drop redundant bang)
     let v = case e of Just x -> Solo x
         x = case v of Solo x -> x
     in v `seq` body

===> (inline seq, float x,y bindings inwards)
     let v = case e of Just x -> Solo x
     in case v of !v -> let x = case v of Solo x -> x
                        in body

===> (fluff up v's pattern; this is a standard Core optimisation)
     let v = case e of Just x -> Solo x
     in case v of v@(Solo p) -> let x = case v of Solo x -> x
                                in body

===> (case of known constructor)
     let v = case e of Just x -> Solo x
     in case v of v@(Solo p) -> let x = p
                                in body

===> (inline x, v)
     case (case e of Just x -> Solo x) of
        Solo p -> body[p/x]

===> (case of case)
     case e of Just x -> body[p/x]
```

The final form is just what we want: a simple case expression. Notice, crucially, that that *pattern* `Just x` is forced eagerly, but `x` itself is not evaluated unless and until `body` does so. Note also that this example uses a pattern that binds exactly one variable, and illustrates the use of the `Solo` 1-tuple.

Rule (SPLIT-STRICT) applies even if the pattern binds no variables:

```
let !(True,False) = e in body

===> (SPLIT-STRICT)
     let v = case e of !(True,False) -> (); () = v in v `seq` body

===> (inline, simplify, drop redundant bang)
     case e of (True,False) -> body
```

That is, we force `e` and check that it has the right form before proceeding with `body`. This happens even if the pattern is itself vacuous:

```
let !_ = e in body

===> (SPLIT-STRICT)
     let v = case e of !_ -> (); () = v in v `seq` body

===> (inline, simplify)
     case e of !_ -> body
```

Again, `e` is forced before evaluating `body`. This (along with `!x = e`) is the reason that (SPLIT-STRICT) uses a bang-pattern in the `case` in the desugared right-hand side.

Note that rule (CASE) applies only when any of the *binders* is unlifted; it is irrelevant whether the binding *itself* is unlifted (see GHC proposal #35). For example (see *Unboxed types and primitive operations* (page 554)):

```
let (# a::Int, b::Bool #) = e in body
===> (SPLIT-LAZY)
    let v = case e of (# a,b #) -> (a,b)
        a = case v of (a,b) -> a
        b = case v of (a,b) -> b
    in body
```

Even though the tuple pattern is unboxed, it is matched only when `a` or `b` are evaluated in `body`.

Here is an example with an unlifted data type:

```
type T :: UnliftedType
data T = MkT Int
f1 x = let MkT y  = blah in body1
f2 x = let z :: T = blah in body2
f3 x = let _ :: T = blah in body3
```

In `f1`, even though `T` is an unlifted type, the pattern `MkT y` binds a lifted variable `y`, so (SPLIT-LAZY) applies, and `blah` is not evaluated until `body1` evaluates `y`. In contrast, in `f2` the pattern `z :: T` binds a variable `z` of unlifted type, so (CASE) applies and the let-binding is strict. In `f3` the pattern binds no variables, so again it is lazy like `f1`.

Here is a recursive case

```
letrec xs :: [Int]  -- Recursive
        !xs = factorial y : xs
in body

===> (SPLIT-STRICT)
     letrec xs = factorial y : xs in xs `seq` body
```

(continues on next page)

(continued from previous page)

```
===> (inline seq)
     letrec xs = factorial y : xs in case xs of xs -> body

===> (eliminate case of value)
     letrec xs = factorial y : xs in body
```

and a polymorphic one:

```
let f :: forall a. [a] -> [a]    -- Polymorphic
    !f = fst (reverse, True)
in body

===> (SPLIT-STRICT)
     let f = /\a. fst (reverse a, True) in f `seq` body

===> (inline seq, inline f)
     case (/\a. fst (reverse a, True)) of !f -> body
```

Notice that the seq is added only in the translation to Core If we did it in Haskell source, thus

```
let f = ... in f `seq` body
```

then f's polymorphic type would get instantiated, so the Core translation would be

```
let f = ... in f Any `seq` body
```

When overloading is involved, the results might be slightly counter intuitive:

```
let f :: forall a. Eq a => a -> [a] -> Bool    -- Overloaded
    !f = fst (member, True)
in body

===> (SPLIT-STRICT)
     let f = /\a \(d::Eq a). fst (member, True) in f `seq` body

===> (inline seq, case of value)
     let f = /\a \(d::Eq a). fst (member, True) in body
```

Note that the bang has no effect at all in this case

6.15 Parallel and Concurrent

GHC implements some major extensions to Haskell to support concurrent and parallel programming. Let us first establish terminology:

- *Parallelism* means running a Haskell program on multiple processors, with the goal of improving performance. Ideally, this should be done invisibly, and with no semantic changes.
- *Concurrency* means implementing a program by using multiple I/O-performing threads. While a concurrent Haskell program *can* run on a parallel machine, the primary goal of using concurrency is not to gain performance, but rather because that is the simplest and most direct way to write the program. Since the threads perform I/O, the semantics of the program is necessarily non-deterministic.

GHC supports both concurrency and parallelism.

6.15.1 Concurrent and Parallel Haskell

6.15.1.1 Concurrent Haskell

Concurrent Haskell is the name given to GHC's concurrency extension. It is enabled by default, so no special flags are required. The Concurrent Haskell paper is still an excellent resource, as is Tackling the awkward squad.

To the programmer, Concurrent Haskell introduces no new language constructs; rather, it appears simply as a library, Control.Concurrent. The functions exported by this library include:

- Forking and killing threads.
- Sleeping.
- Synchronised mutable variables, called `MVars`
- Support for bound threads; see the paper Extending the FFI with concurrency.

6.15.1.2 Parallel Haskell

GHC includes support for running Haskell programs in parallel on symmetric, shared-memory multi-processor (SMP). By default GHC runs your program on one processor; if you want it to run in parallel you must link your program with the *-threaded* (page 248), and run it with the RTS *-N ⟨x⟩* (page 132) option; see *Using SMP parallelism* (page 132)). The runtime will schedule the running Haskell threads among the available OS threads, running as many in parallel as you specified with the *-N ⟨x⟩* (page 132) RTS option.

6.15.1.3 Annotating pure code for parallelism

Ordinary single-threaded Haskell programs will not benefit from enabling SMP parallelism alone: you must expose parallelism to the compiler. One way to do so is forking threads using Concurrent Haskell (*Concurrent Haskell* (page 550)), but the simplest mechanism for extracting parallelism from pure code is to use the `par` combinator, which is closely related to (and often used with) `seq`. Both of these are available from the parallel library:

```
infixr 0 `par`
infixr 1 `pseq`

par  :: a -> b -> b
pseq :: a -> b -> b
```

The expression ``(x `par` y)`` *sparks* the evaluation of `x` (to weak head normal form) and returns `y`. Sparks are queued for execution in FIFO order, but are not executed immediately. If the runtime detects that there is an idle CPU, then it may convert a spark into a real thread, and run the new thread on the idle CPU. In this way the available parallelism is spread amongst the real CPUs.

For example, consider the following parallel version of our old nemesis, `nfib`:

```
import Control.Parallel

nfib :: Int -> Int
nfib n | n <= 1 = 1
       | otherwise = par n1 (pseq n2 (n1 + n2))
                     where n1 = nfib (n-1)
                           n2 = nfib (n-2)
```

For values of `n` greater than 1, we use `par` to spark a thread to evaluate `nfib (n-1)`, and then we use `pseq` to force the parent thread to evaluate `nfib (n-2)` before going on to add together these two subexpressions. In this divide-and-conquer approach, we only spark a new thread for one branch of the computation (leaving the parent to evaluate the other branch). Also, we must use `pseq` to ensure that the parent will evaluate `n2` *before* `n1` in the expression `(n1 + n2 + 1)`. It is not sufficient to reorder the expression as `(n2 + n1 + 1)`, because the compiler may not generate code to evaluate the addends from left to right.

Note that we use `pseq` rather than `seq`. The two are almost equivalent, but differ in their runtime behaviour in a subtle way: `seq` can evaluate its arguments in either order, but `pseq` is required to evaluate its first argument before its second, which makes it more suitable for controlling the evaluation order in conjunction with `par`.

When using `par`, the general rule of thumb is that the sparked computation should be required at a later time, but not too soon. Also, the sparked computation should not be too small, otherwise the cost of forking it in parallel will be too large relative to the amount of parallelism gained. Getting these factors right is tricky in practice.

It is possible to glean a little information about how well `par` is working from the runtime statistics; see *RTS options to control the garbage collector* (page 184).

More sophisticated combinators for expressing parallelism are available from the `Control.Parallel.Strategies` module in the parallel package. This module builds functionality around `par`, expressing more elaborate patterns of parallel computation, such as parallel `map`.

6.15.2 Software Transactional Memory

GHC now supports a new way to coordinate the activities of Concurrent Haskell threads, called Software Transactional Memory (STM). The STM papers are an excellent introduction to what STM is, and how to use it.

The main library you need to use is the stm library. The main features supported are these:

- Atomic blocks.
- Transactional variables.
- Operations for composing transactions: `retry`, and `orElse`.
- Data invariants.

All these features are described in the papers mentioned earlier.

6.15.3 Static pointers

StaticPointers

Since 7.10.1

Allow use of static pointer syntax.

The language extension *StaticPointers* (page 552) adds a new syntactic form `static e`, which stands for a reference to the closed expression ⟨e⟩. This reference is stable and portable, in the sense that it remains valid across different processes on possibly different machines. Thus, a process can create a reference and send it to another process that can resolve it to ⟨e⟩.

With this extension turned on, `static` is no longer a valid identifier.

Static pointers were first proposed in the paper Towards Haskell in the cloud, Jeff Epstein, Andrew P. Black and Simon Peyton-Jones, Proceedings of the 4th ACM Symposium on Haskell, pp. 118-129, ACM, 2011.

6.15.3.1 Using static pointers

Each reference is given a key which can be used to locate it at runtime with GHC.StaticPtr.unsafeLookupStaticPtr which uses a global and immutable table called the Static Pointer Table. The compiler includes entries in this table for all static forms found in the linked modules. The value can be obtained from the reference via GHC.StaticPtr.deRefStaticPtr.

The body `e` of a `static e` expression must be a closed expression. Where we say an expression is *closed* when all of its free (type) variables are closed. And a variable is *closed* if it is let-bound to a *closed* expression and its type is *closed* as well. And a type is *closed* if it has no free variables.

All of the following are permissible:

```
inc :: Int -> Int
inc x = x + 1

ref1 = static 'a'
ref2 = static inc
ref3 = static (inc 1)
ref4 = static ((\x -> x + 1) (1 :: Int))
ref5 = static (let x = 'a' in x)
ref6 = let x = 'a' in static x
```

While the following definitions are rejected:

```
ref7 y = let x = y in static x    -- x is not closed
ref8 y = static (let x = 1 in y)  -- y is not let-bound
ref9 (y :: a) = let x = undefined :: a
                in static x      -- x has a non-closed type
```

Note: While modules loaded in GHCi with the *:load* (page 50) command may use *StaticPointers* (page 552) and `static` expressions, statements entered on the REPL may not. This is a limitation of GHCi; see #12356 for details.

Note: The set of keys used for locating static pointers in the Static Pointer Table is not guaranteed to remain stable for different program binaries. Or in other words, only processes launched from the same program binary are guaranteed to use the same set of keys.

6.15.3.2 Static semantics of static pointers

Informally, if we have a closed expression

```
e :: forall a_1 ... a_n . t
```

the static form is of type

```
static e :: (IsStatic p, Typeable a_1, ... , Typeable a_n) => p t
```

A static form determines a value of type `StaticPtr t`, but just like `OverloadedLists` and `OverloadedStrings`, this literal expression is overloaded to allow lifting a `StaticPtr` into another type implicitly, via the `IsStatic` class:

```
class IsStatic p where
    fromStaticPtr :: Typeable a => StaticPtr a -> p a
```

The only predefined instance is the obvious one that does nothing:

```
instance IsStatic StaticPtr where
    fromStaticPtr sptr = sptr
```

See GHC.StaticPtr.IsStatic.

Furthermore, type `t` is constrained to have a `Typeable` instance. The following are therefore illegal:

```
static id                      -- No Typeable instance for (a -> a)
static Control.Monad.ST.runST  -- No Typeable instance for ((forall s. ST s a) -> a)
```

That being said, with the appropriate use of wrapper datatypes, the above limitations induce no loss of generality:

```
{-# LANGUAGE ConstraintKinds           #-}
{-# LANGUAGE ExistentialQuantification #-}
{-# LANGUAGE Rank2Types                #-}
{-# LANGUAGE StandaloneDeriving        #-}
{-# LANGUAGE StaticPointers            #-}

import Control.Monad.ST
import Data.Typeable
import GHC.StaticPtr

data Dict c = c => Dict

g1 :: Typeable a => StaticPtr (Dict (Show a) -> a -> String)
g1 = static (\Dict -> show)

data Rank2Wrapper f = R2W (forall s. f s)
  deriving Typeable
```

(continues on next page)

(continued from previous page)

```
newtype Flip f a s = Flip { unFlip :: f s a }
  deriving Typeable

g2 :: Typeable a => StaticPtr (Rank2Wrapper (Flip ST a) -> a)
g2 = static (\(R2W f) -> runST (unFlip f))
```

6.16 Unboxed types and primitive operations

GHC is built on a raft of primitive data types and operations; "primitive" in the sense that they cannot be defined in Haskell itself. While you really can use this stuff to write fast code, we generally find it a lot less painful, and more satisfying in the long run, to use higher-level language features and libraries. With any luck, the code you write will be optimised to the efficient unboxed version in any case. And if it isn't, we'd like to know about it.

All these primitive data types and operations are exported by the module GHC.Exts.

If you want to mention any of the primitive data types or operations in your program, you must first import `GHC.Exts` to bring them into scope. Many of them have names ending in `#`, and to mention such names you need the *MagicHash* (page 278) extension.

To enable defining literals for other primitive data types, see the *ExtendedLiterals* (page 493) extension.

The primops make extensive use of *unboxed types* (page 554) and *unboxed tuples* (page 556), which we briefly summarise here.

6.16.1 Unboxed types

Most types in GHC are boxed, which means that values of that type are represented by a pointer to a heap object. The representation of a Haskell `Int`, for example, is a two-word heap object. An unboxed type, however, is represented by the value itself, no pointers or heap allocation are involved.

Unboxed types correspond to the "raw machine" types you would use in C: `Int#` (long int), `Double#` (double), `Addr#` (void *), etc. The *primitive operations* (PrimOps) on these types are what you might expect; e.g., `(+#)` is addition on `Int#`s, and is the machine-addition that we all know and love—usually one instruction.

Primitive (unboxed) types cannot be defined in Haskell, and are therefore built into the language and compiler. Primitive types are always unlifted; that is, a value of a primitive type cannot be bottom. (Note: a "boxed" type means that a value is represented by a pointer to a heap object; a "lifted" type means that terms of that type may be bottom. See the next paragraph for an example.) We use the convention (but it is only a convention) that primitive types, values, and operations have a # suffix (see *The magic hash* (page 278)). For some primitive types we have special syntax for literals, also described in the *same section* (page **??**).

Primitive values are often represented by a simple bit-pattern, such as `Int#`, `Float#`, `Double#`. But this is not necessarily the case: a primitive value might be represented by a pointer to a heap-allocated object. Examples include `Array#`, the type of primitive arrays. Thus, `Array#` is an unlifted, boxed type. A primitive array is heap-allocated because it is too big a value to fit in a register, and would be too expensive to copy around; in a sense, it is accidental that it is represented by a pointer. If a pointer represents a primitive value, then it really does point to that value: no unevaluated thunks, no indirections. Nothing can be at the other end of the pointer than the primitive value. A numerically-intensive program using unboxed types can go a *lot* faster than its "standard" counterpart—we saw a threefold speedup on one example.

6.16.2 Unboxed type kinds

Because unboxed types are represented without the use of pointers, we cannot store them in a polymorphic data type. For example, the `Just` node of `Just 42#` would have to be different from the `Just` node of `Just 42`; the former stores an integer directly, while the latter stores a pointer. GHC currently does not support this variety of `Just` nodes (nor for any other data type). Accordingly, the *kind* of an unboxed type is different from the kind of a boxed type.

The Haskell Report describes that `*` (spelled `Type` and imported from `Data.Kind` in the GHC dialect of Haskell) is the kind of ordinary data types, such as `Int`. Furthermore, type constructors can have kinds with arrows; for example, `Maybe` has kind `Type -> Type`. Unboxed types have a kind that specifies their runtime representation. For example, the type `Int#` has kind `TYPE IntRep` and `Double#` has kind `TYPE DoubleRep`. These kinds say that the runtime representation of an `Int#` is a machine integer, and the runtime representation of a `Double#` is a machine double-precision floating point. In contrast, the kind `Type` is actually just a synonym for `TYPE LiftedRep`. More details of the `TYPE` mechanisms appear in the *section on runtime representation polymorphism* (page **??**).

Given that `Int#`'s kind is not `Type`, then it follows that `Maybe Int#` is disallowed. Similarly, because type variables tend to be of kind `Type` (for example, in `(.) :: (b -> c) -> (a -> b) -> a -> c`, all the type variables have kind `Type`), polymorphism tends not to work over primitive types. Stepping back, this makes some sense, because a polymorphic function needs to manipulate the pointers to its data, and most primitive types are unboxed.

There are some restrictions on the use of primitive types:

- You cannot define a newtype whose representation type (the argument type of the data constructor) is an unboxed type. Thus, this is illegal:

  ```
  newtype A = MkA Int#
  ```

  However, this restriction can be relaxed by enabling *`UnliftedNewtypes`* (page 558). The *section on unlifted newtypes* (page 558) details the behavior of such types.

- You cannot bind a variable with an unboxed type in a *top-level* binding.
- You cannot bind a variable with an unboxed type in a *recursive* binding.
- You may bind unboxed variables in a (non-recursive, non-top-level) pattern binding, but you must make any such pattern-match strict. (Failing to do so emits a warning *`-Wunbanged-strict-patterns`* (page 106).) For example, rather than:

  ```
  data Foo = Foo Int Int#

  f x = let (Foo a b, w) = ..rhs.. in ..body..
  ```

  you must write:

  ```
  data Foo = Foo Int Int#

  f x = let !(Foo a b, w) = ..rhs.. in ..body..
  ```

  since `b` has type `Int#`. See *Dynamic semantics of bang patterns* (page 545).

6.16.3 Unboxed tuples

UnboxedTuples

> **Implies** *UnboxedSums* (page 557)
>
> **Since** 6.8.1

Unboxed tuples aren't really exported by `GHC.Exts`; they are a syntactic extension (*UnboxedTuples* (page 556)). An unboxed tuple looks like this:

```
(# e_1, ..., e_n #)
```

where `e_1..e_n` are expressions of any type (primitive or non-primitive). The type of an unboxed tuple looks the same.

Note that when unboxed tuples are enabled, `(#` is a single lexeme, so for example when using operators like `#` and `#-` you need to write `( # )` and `( #- )` rather than `(#)` and `(#-)`.

Unboxed tuples are used for functions that need to return multiple values, but they avoid the heap allocation normally associated with using fully-fledged tuples. When an unboxed tuple is returned, the components are put directly into registers or on the stack; the unboxed tuple itself does not have a composite representation. Many of the primitive operations listed in `primops.txt.pp` return unboxed tuples. In particular, the `IO` and `ST` monads use unboxed tuples to avoid unnecessary allocation during sequences of operations.

The typical use of unboxed tuples is simply to return multiple values, binding those multiple results with a `case` expression, thus:

```
f x y = (# x+1, y-1 #)
g x = case f x x of { (# a, b #) -> a + b }
```

You can have an unboxed tuple in a pattern binding, thus

```
f x = let (# p,q #) = h x in ..body..
```

If the types of `p` and `q` are not unboxed, the resulting binding is lazy like any other Haskell pattern binding. The above example desugars like this:

```
f x = let t = case h x of { (# p,q #) -> (p,q) }
          p = fst t
          q = snd t
      in ..body..
```

Indeed, the bindings can even be recursive. See *Dynamic semantics of bang patterns* (page 545) for a more precise account.

To refer to the unboxed tuple type constructors themselves, e.g. if you want to attach instances to them, use `(# #)`, `(#,#)`, `(#,,#)`, etc. This mirrors the syntax for boxed tuples `()`, `(,)`, `(,,)`, etc.

6.16.4 Unboxed sums

UnboxedSums

Implied by *UnboxedTuples* (page 556)

Since 8.2.1

Enable the use of unboxed sum syntax. Implied by *UnboxedTuples* (page 556).

-XUnboxedSums enables new syntax for anonymous, unboxed sum types. The syntax for an unboxed sum type with N alternatives is

```
(# t_1 | t_2 | ... | t_N #)
```

where `t_1` … `t_N` are types (which can be unlifted, including unboxed tuples and sums).

Unboxed tuples can be used for multi-arity alternatives. For example:

```
(# (# Int, String #) | Bool #)
```

The term level syntax is similar. Leading and preceding bars (|) indicate which alternative it is. Here are two terms of the type shown above:

```
(# (# 1, "foo" #) | #) -- first alternative

(# | True #) -- second alternative
```

The pattern syntax reflects the term syntax:

```
case x of
  (# (# i, str #) | #) -> ...
  (# | bool #) -> ...
```

Note that spaces are always required around bars. For example, `(# | 1# | | #)` is valid, but `(# | 1# || #)` and `(#| 1# | | #)` are both invalid.

The type constructors themselves can be written in prefix form as `(# | #)`, `(# | | #)`, `(# | | | #)`, etc. Partial applications must also use prefix form, i.e. `(# | #) Int#`. Saturated applications can be written either way, so that `(# | #) Int# Float#` is equivalent to `(# Int# | Float# #)`.

Unboxed sums are "unboxed" in the sense that, instead of allocating sums in the heap and representing values as pointers, unboxed sums are represented as their components, just like unboxed tuples. These "components" depend on alternatives of a sum type. Like unboxed tuples, unboxed sums are lazy in their lifted components.

The code generator tries to generate as compact layout as possible for each unboxed sum. In the best case, size of an unboxed sum is size of its biggest alternative plus one word (for a tag). The algorithm for generating the memory layout for a sum type works like this:

- All types are classified as one of these classes: 32bit word, 64bit word, 32bit float, 64bit float, pointer.
- For each alternative of the sum type, a layout that consists of these fields is generated. For example, if an alternative has `Int`, `Float#` and `String` fields, the layout will have an 32bit word, 32bit float and pointer fields.
- Layout fields are then overlapped so that the final layout will be as compact as possible. For example, suppose we have the unboxed sum:

```
(# (# Word32#, String, Float# #)
|  (# Float#, Float#, Maybe Int #) #)
```

The final layout will be something like

```
Int32, Float32, Float32, Word32, Pointer
```

The first `Int32` is for the tag. There are two `Float32` fields because floating point types can't overlap with other types, because of limitations of the code generator that we're hoping to overcome in the future. The second alternative needs two `Float32` fields: The `Word32` field is for the `Word32#` in the first alternative. The `Pointer` field is shared between `String` and `Maybe Int` values of the alternatives.

As another example, this is the layout for the unboxed version of `Maybe a` type, `(# (# #) | a #)`:

```
Int32, Pointer
```

The `Pointer` field is not used when tag says that it's `Nothing`. Otherwise `Pointer` points to the value in `Just`. As mentioned above, this type is lazy in its lifted field. Therefore, the type

```
data Maybe' a = Maybe' (# (# #) | a #)
```

is *precisely* isomorphic to the type `Maybe a`, although its memory representation is different.

In the degenerate case where all the alternatives have zero width, such as the `Bool`-like `(# (# #) | (# #) #)`, the unboxed sum layout only has an `Int32` tag field (i.e., the whole thing is represented by an integer).

6.16.5 Unlifted Newtypes

UnliftedNewtypes

Since 8.10.1

Enable the use of newtypes over types with non-lifted runtime representations.

GHC implements an *UnliftedNewtypes* (page 558) extension as specified in the GHC proposal #98. *UnliftedNewtypes* (page 558) relaxes the restrictions around what types can appear inside of a `newtype`. For example, the type

```
newtype A = MkA Int#
```

is accepted when this extension is enabled. This creates a type `A :: TYPE IntRep` and a data constructor `MkA :: Int# -> A`. Although the kind of `A` is inferred by GHC, there is nothing visually distinctive about this type that indicated that is it not of kind `Type` like newtypes typically are. *GADTSyntax* (page **??**) can be used to provide a kind signature for additional clarity

```
newtype A :: TYPE IntRep where
  MkA :: Int# -> A
```

The `Coercible` machinery works with unlifted newtypes just like it does with lifted types. In either of the equivalent formulations of `A` given above, users would additionally have access to a coercion between `A` and `Int#`.

As a consequence of the *representation-polymorphic binder restriction* (page **??**), representation-polymorphic fields are disallowed in data constructors of data types declared using `data`. However, since `newtype` data constructor application is implemented as a coercion instead of as function application, this restriction does not apply to the field inside a `newtype` data constructor. Thus, the type checker accepts

```
newtype Identity# :: forall (r :: RuntimeRep). TYPE r -> TYPE r where
  MkIdentity# :: forall (r :: RuntimeRep) (a :: TYPE r). a -> Identity# a
```

And with *UnboxedSums* (page 557) enabled

```
newtype Maybe# :: forall (r :: RuntimeRep). TYPE r -> TYPE (SumRep '[r, TupleRep
↪'[]]) where
  MkMaybe# :: forall (r :: RuntimeRep) (a :: TYPE r). (# a | (# #) #) -> Maybe# a
```

This extension also relaxes some of the restrictions around data family instances. In particular, *UnliftedNewtypes* (page 558) permits a `newtype instance` to be given a return kind of `TYPE r`, not just `Type`. For example, the following `newtype instance` declarations would be permitted:

```
class Foo a where
  data FooKey a :: TYPE IntRep
class Bar (r :: RuntimeRep) where
  data BarType r :: TYPE r

instance Foo Bool where
  newtype FooKey Bool = FooKeyBoolC Int#
instance Bar WordRep where
  newtype BarType WordRep = BarTypeWordRepC Word#
```

It is worth noting that *UnliftedNewtypes* (page 558) is *not* required to give the data families themselves return kinds involving `TYPE`, such as the `FooKey` and `BarType` examples above. The extension is only required for `newtype instance` declarations, such as `FooKeyBoolC` and `BarTypeWorkRepC` above.

This extension impacts the determination of whether or not a newtype has a Complete User-Supplied Kind (CUSK). The exact impact is specified *the section on CUSKs* (page **??**).

6.16.6 Unlifted Datatypes

UnliftedDatatypes

> **Implies** *DataKinds* (page 357), *StandaloneKindSignatures* (page 369)
>
> **Since** 9.2.1
>
> Enable the declaration of data types with unlifted or levity-polymorphic result kind.

GHC implements the *UnliftedDatatypes* (page 559) extension as specified in the GHC proposal #265. *UnliftedDatatypes* (page 559) relaxes the restrictions around what result kinds are allowed in data declarations. For example, the type

```
data UList a :: UnliftedType where
  UCons :: a -> UList a -> UList a
  UNil :: UList a
```

defines a list type that lives in kind `UnliftedType` (e.g., `TYPE (BoxedRep Unlifted)`). As such, each occurrence of a term of that type is assumed to be evaluated (and the compiler makes sure that is indeed the case). In other words: Unlifted data types behave like data types in strict languages such as OCaml or Idris. However unlike *StrictData* (page 542), this extension will not change whether the fields of a (perhaps unlifted) data type are strict or lazy. For example, UCons is lazy in its first argument as its field has kind Type.

The fact that unlifted types are always evaluated allows GHC to elide evaluatedness checks at runtime. See the Motivation section of the proposal for how this can improve performance for some programs.

The above data declaration in GADT syntax correctly suggests that unlifted data types are compatible with the full GADT feature set. Somewhat conversely, you can also declare unlifted data types in Haskell98 syntax, which requires you to specify the result kind via *StandaloneKindSignatures* (page 369):

```
type UList :: Type -> UnliftedType
data UList a = UCons a (UList a) | UNil
```

You may even declare levity-polymorphic data types:

```
type PEither :: Type -> Type -> TYPE (BoxedRep l)
data PEither l r = PLeft l | PRight r

f :: PEither @Unlifted Int Bool -> Bool
f (PRight b) = b
f _          = False
```

While f above could reasonably be levity-polymorphic (as it evaluates its argument either way), GHC currently disallows the more general type PEither @l Int Bool -> Bool. This is a consequence of the *representation-polymorphic binder restriction* (page **??**).

Pattern matching against an unlifted data type work just like that for lifted types; but see *Dynamic semantics of bang patterns* (page 545) for the semantics of pattern bindings involving unlifted data types.

Due to #19487, it is not currently possible to declare levity-polymorphic data types with nullary data constructors. There is a workaround, though:

```
type T :: TYPE (BoxedRep l)
data T where
  MkT :: forall l. (() :: Constraint) => T @l
```

The use of => makes the type of MkT lifted. If you want a zero-runtime-cost alternative, use MkT :: Proxy# () -> T @l instead and bear with the additional proxy# argument at construction sites.

This extension also relaxes some of the restrictions around data family instances. In particular, *UnliftedDatatypes* (page 559) permits a data instance to be given a return kind that unifies with TYPE (BoxedRep l), not just Type. For example, the following data instance declarations would be permitted:

```
data family F a :: UnliftedType
data instance F Int = FInt

data family G a :: TYPE (BoxedRep l)
data instance G Int = GInt Int -- defaults to Type
data instance G Bool :: UnliftedType where
  GBool :: Bool -> G Bool
data instance G Char :: Type where
```

(continues on next page)

(continued from previous page)

```
  GChar :: Char -> G Char
data instance G Double :: forall l. TYPE (BoxedRep l) where
  GDouble :: Int -> G @l Double
```

It is worth noting that *UnliftedDatatypes* (page 559) is *not* required to give the data families themselves return kinds involving `TYPE`, such as the `G` example above. The extension is only required for `data instance` declarations, such as `FInt` and `GBool` above.

6.17 Foreign function interface (FFI)

ForeignFunctionInterface

Since 6.8.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271), *Haskell2010* (page 272)

Allow use of the Haskell foreign function interface.

GHC (mostly) conforms to the Haskell Foreign Function Interface as specified in the Haskell Report. Refer to the relevant chapter of the Haskell Report for more details.

FFI support is enabled by default, but can be enabled or disabled explicitly with the *ForeignFunctionInterface* (page 561) flag.

GHC implements a number of GHC-specific extensions to the FFI Chapter of the Haskell 2010 Report. These extensions are described in *GHC extensions to the FFI Chapter* (page 562), but please note that programs using these features are not portable. Hence, these features should be avoided where possible.

The FFI libraries are documented in the accompanying library documentation; see for example the Foreign module.

6.17.1 GHC differences to the FFI Chapter

6.17.1.1 Guaranteed call safety

The Haskell 2010 Report specifies that `safe` FFI calls must allow foreign calls to safely call into Haskell code. In practice, this means that called functions also have to assume heap-allocated Haskell values may move around arbitrarily in order to allow for GC.

This greatly constrains library authors since it implies that it is not safe to pass any heap object reference to a `safe` foreign function call. For instance, it is often desirable to pass *unpinned* (page 577) `ByteArray#`s directly to native code to avoid making an otherwise-unnecessary copy. However, this can not be done safely for `safe` calls since the array might be moved by the garbage collector in the middle of the call.

The Chapter *does* allow for implementations to move objects around during `unsafe` calls as well. So strictly Haskell 2010-conforming programs cannot pass heap-allocated references to `unsafe` FFI calls either.

GHC, since version 8.4, **guarantees** that garbage collection will never occur during an `unsafe` call, even in the bytecode interpreter, and further guarantees that `unsafe` calls will be performed in the calling thread. Making it safe to pass heap-allocated objects to unsafe functions.

In previous releases, GHC would take advantage of the freedom afforded by the Chapter by performing `safe` foreign calls in place of `unsafe` calls in the bytecode interpreter. This meant that some packages which worked when compiled would fail under GHCi (e.g. #13730). But this is no longer the case in recent releases.

6.17.1.2 Interactions between `safe` calls and bound threads

A `safe` call calling into haskell is run on a bound thread by the RTS. This means any nesting of `safe` calls will be executed on the same operating system thread. *Sequential* `safe` calls however do not enjoy this luxury and may be run on arbitrary OS threads.

This behaviour is considered an implementation detail and code relying on thread local state should instead use one of the interfaces provided in Control.Concurrent to make this explicit.

For information on what bound threads are, see the documentation for the Control.Concurrent.

For more details on the implementation see the Paper: "Extending the Haskell Foreign Function Interface with Concurrency". Last known to be accessible here.

6.17.1.3 Varargs not supported by `ccall` calling convention

Note that functions requiring varargs arguments are unsupported by the `ccall` calling convention. Foreign imports needing to call such functions should rather use the `capi` convention, giving an explicit signature for the needed call-pattern. For instance, one could write:

```
foreign import "capi" "printf"
    my_printf :: Ptr CChar -> CInt -> IO ()

printInt :: CInt -> IO ()
printInt n = my_printf "printed number %d" n
```

6.17.2 GHC extensions to the FFI Chapter

The FFI features that are described in this section are specific to GHC. Your code will not be portable to other compilers if you use them.

6.17.2.1 Unlifted FFI Types

`UnliftedFFITypes`

Since 6.8.1

The following unlifted unboxed types may be used as basic foreign types (see FFI Chapter, Section 8.6) for both `safe` and `unsafe` foreign calls: `Int#`, `Word#`, `Char#`, `Float#`, `Double#`, `Addr#`, and `StablePtr# a`. Several unlifted boxed types may be used as arguments to FFI calls, subject to these restrictions:

- Valid arguments for `foreign import unsafe` FFI calls: `Array#`, `SmallArray#`, `ByteArray#`, and the mutable counterparts of these types.
- Valid arguments for `foreign import safe` FFI calls: `ByteArray#` and `MutableByteArray#`. The byte array must be *pinned* (page 577).

- Mutation: In both `foreign import unsafe` and `foreign import safe` FFI calls, it is safe to mutate a `MutableByteArray`. Mutating any other type of array leads to undefined behavior. Reason: Mutable arrays of heap objects record writes for the purpose of garbage collection. An array of heap objects is passed to a foreign C function, the runtime does not record any writes. Consequently, it is not safe to write to an array of heap objects in a foreign function. Since the runtime has no facilities for tracking mutation of a `MutableByteArray#`, these can be safely mutated in any foreign function.
- Note that `safe` FFI calls don't take any measures to keep their arguments alive while the called C function runs. For arguments who's live time doesn't extend past the FFI call `keepAlive#` or a `StablePtr` should be used to ensure the argument isn't garbage collected before the call finishes.

None of these restrictions are enforced at compile time. Failure to heed these restrictions will lead to runtime errors that can be very difficult to track down. (The errors likely will not manifest until garbage collection happens.) In tabular form, these restrictions are:

Table 2: Restrictions on unlifted boxed arguments passed to foreign C calls. Cells marked as "Unsound" represent combinations that lead to undefined runtime behavior. GHC does not reject such unsound programs at compile time.

	When value is used as argument to FFI call that is			
	`foreign import safe`		`foreign import unsafe`	
Argument Type	reads are	writes are	reads are	writes are
`Array#`	Unsound	Unsound	Sound	Unsound
`MutableArray#`	Unsound	Unsound	Sound	Unsound
`SmallArray#`	Unsound	Unsound	Sound	Unsound
`MutableSmallArray#`	Unsound	Unsound	Sound	Unsound
unpinned `ByteArray#`	Unsound	Unsound	Sound	Unsound
unpinned `MutableByteArray#`	Unsound	Unsound	Sound	Sound
pinned `ByteArray#`	Sound	Unsound	Sound	Unsound
pinned `MutableByteArray#`	Sound	Sound	Sound	Sound

When passing any of the unlifted array types as an argument to a foreign C call, a foreign function sees a pointer that refers to the payload of the array, not to the `StgArrBytes`/`StgMutArrPtrs`/`StgSmallMutArrPtrs` heap object containing it[1]. By contrast, a *foreign Cmm call* (page 565), introduced by `foreign import prim`, sees the heap object, not just the payload. This means that, in some situations, the foreign C function might not need any knowledge of the RTS closure types. The following example sums the first three bytes in a `MutableByteArray#`[2] without using anything from `Rts.h`:

```
// C source
uint8_t add_triplet(uint8_t* arr) {
  return (arr[0] + arr[1] + arr[2]);
}

-- Haskell source
foreign import ccall unsafe "add_triplet"
  addTriplet :: MutableByteArray# RealWorld -> IO Word8
```

[1] Prior to GHC 8.10, when passing an `ArrayArray#` argument to a foreign function, the foreign function would see a pointer to the `StgMutArrPtrs` rather than just the payload.

[2] In practice, the FFI should not be used for a task as simple as reading bytes from a `MutableByteArray#`. Users should prefer `GHC.Exts.readWord8Array#` for this.

In other situations, the C function may need knowledge of the RTS closure types. The following example sums the first element of each `ByteArray#` (interpreting the bytes as an array of `CInt`) element of an `Array# ByteArray#`[3]:

```
// C source, must include the RTS to make the struct StgArrBytes
// available along with its fields, such as `payload`.
#include "Rts.h"
int sum_first (StgArrBytes **bufs, StgWord sz) {
  int res = 0;
  for(StgWord ix = 0; ix < sz; ix++) {
    res = res + ((int*)(bufs[ix]->payload))[0];
  }
  return res;
}

-- Haskell source
foreign import ccall unsafe "sum_first"
  sumFirst :: Array# ByteArray# -> CInt -> IO CInt

sumFirst' :: Array# ByteArray# -> IO CInt
sumFirst' arr = sumFirst arr (sizeofArray# arr)
```

Although GHC allows the user to pass all unlifted boxed types to foreign functions, some of them are not amenable to useful work. Although `Array#` is unlifted, the elements in its payload can be lifted, and a foreign C function cannot safely force thunks. Consequently, a foreign C function may not dereference any of the addresses that comprise the payload of `Array# a` if `a` has a lifted representation.

6.17.2.2 Newtype wrapping of the IO monad

The FFI spec requires the IO monad to appear in various places, but it can sometimes be convenient to wrap the IO monad in a `newtype`, thus:

```
newtype MyIO a = MIO (IO a)
```

(A reason for doing so might be to prevent the programmer from calling arbitrary IO procedures in some part of the program.)

The Haskell FFI already specifies that arguments and results of foreign imports and exports will be automatically unwrapped if they are newtypes (Section 3.2 of the FFI addendum). GHC extends the FFI by automatically unwrapping any newtypes that wrap the IO monad itself. More precisely, wherever the FFI specification requires an `IO` type, GHC will accept any newtype-wrapping of an `IO` type. For example, these declarations are OK:

```
foreign import foo :: Int -> MyIO Int
foreign import "dynamic" baz :: (Int -> MyIO Int) -> CInt -> MyIO Int
```

[3] As in[2], the FFI is not actually needed for this. `GHC.Exts` includes primitives for reading from an `Array# a`, such as `GHC.Exts.indexArray#`.

6.17.2.3 Explicit "forall"s in foreign types

The type variables in the type of a foreign declaration may be quantified with an explicit `forall` by using the *ExplicitForAll* (page 506) language extension, as in the following example:

```
{-# LANGUAGE ExplicitForAll #-}
foreign import ccall "mmap" c_mmap :: forall a. CSize -> IO (Ptr a)
```

Note that an explicit `forall` must appear at the front of the type signature and is not permitted to appear nested within the type, as in the following (erroneous) examples:

```
foreign import ccall "mmap" c_mmap' :: CSize -> forall a. IO (Ptr a)
foreign import ccall quux :: (forall a. Ptr a) -> IO ()
```

6.17.2.4 Primitive imports

GHCForeignImportPrim

Since 6.12.1

Status InternalUseOnly

With *GHCForeignImportPrim* (page 565), GHC extends the FFI with an additional calling convention `prim`, e.g.:

```
foreign import prim "foo" foo :: ByteArray# -> (# Int#, Int# #)
```

This is used to import functions written in Cmm code that follow an internal GHC calling convention. The arguments and results must be unboxed types, except that an argument may be of type `Any :: Type` or `Any :: UnliftedType` (which can be arranged by way of `unsafeCoerce#`) and the result type is allowed to be an unboxed tuple or the types `Any :: Type` or `Any :: UnliftedType`.

This feature is not intended for use outside of the core libraries that come with GHC. For more details see the GHC developer wiki.

6.17.2.5 Interruptible foreign calls

InterruptibleFFI

Since 7.2.1

This concerns the interaction of foreign calls with `Control.Concurrent.throwTo`. Normally when the target of a `throwTo` is involved in a foreign call, the exception is not raised until the call returns, and in the meantime the caller is blocked. This can result in unresponsiveness, which is particularly undesirable in the case of user interrupt (e.g. Control-C). The default behaviour when a Control-C signal is received (`SIGINT` on Unix) is to raise the `UserInterrupt` exception in the main thread; if the main thread is blocked in a foreign call at the time, then the program will not respond to the user interrupt.

The problem is that it is not possible in general to interrupt a foreign call safely. However, GHC does provide a way to interrupt blocking *system* calls which works for most system calls on both Unix and Windows.

When the `InterruptibleFFI` extension is enabled, a foreign call can be annotated with `interruptible` instead of `safe` or `unsafe`:

```
foreign import ccall interruptible
   "sleep" sleepBlock :: CUint -> IO CUint
```

`interruptible` behaves exactly as `safe`, except that when a `throwTo` is directed at a thread in an interruptible foreign call, irrespective of the masking state, the exception is added to the blocked exceptions queue of the target thread and an OS-specific mechanism will be used to attempt to cause the foreign call to return:

Unix systems The thread making the foreign call is sent a `SIGPIPE` signal using `pthread_kill()`. This is usually enough to cause a blocking system call to return with `EINTR` (GHC by default installs an empty signal handler for `SIGPIPE`, to override the default behaviour which is to terminate the process immediately).

Windows systems [Vista and later only] The RTS calls the Win32 function `CancelSynchronousIo`, which will cause a blocking I/O operation to return with the error `ERROR_OPERATION_ABORTED`.

Once the system call is successfully interrupted, the surrounding code must return control out of the `foreign import`, back into Haskell code, so that any blocked exception can be raised if the masking state of the thread allows it. Being under mask gives the Haskell code an opportunity to detect and react to the interrupt error code from the c call.

If the foreign code simply retries the system call directly without returning back to Haskell, then the intended effect of *interruptible* disappears and functions like System.Timeout.timeout will not work.

Finally, after the `interruptible` foreign call returns into Haskell, the Haskell code should allow exceptions to be raised (`Control.Exception`'s `allowInterrupt`, or `interruptible yield` for non-`-threaded`, see #8684), and implement the `EINTR`-retrying in Haskell (e.g. using e.g. Foreign.C.Error.throwErrnoIfMinus1Retry).

Be especially careful when using `interruptible` to check that the called foreign function is prepared to deal with the consequences of the call being interrupted. On Unix it is considered good practice to always check for `EINTR` after system calls, so you can expect it not to crash (but in that case `interruptible` will not work as intended unless the code then returns all the way up to Haskell as described above). But on Windows it is not typically common practice to handle `ERROR_OPERATION_ABORTED`.

The approach works *only* for foreign code that does I/O (system calls), not for CPU-intensive computations that do not do any system calls. This is because the only way by which the foreign code can observe interruption is by system calls returning interruption error codes. To be able to interrupt long-running foreign code doing no system calls, the code must likely be changed to explicitly check for intended early termination.

6.17.2.6 The CAPI calling convention

CApiFFI

Since 7.6.1

The `CApiFFI` extension allows a calling convention of `capi` to be used in foreign declarations, e.g.

```
foreign import capi "header.h f" f :: CInt -> IO CInt
```

Rather than generating code to call `f` according to the platform's ABI, we instead call `f` using the C API defined in the header `header.h`. Thus `f` can be called even if it may be defined as a CPP `#define` rather than a proper function.

When using `capi`, it is also possible to import values, rather than functions. For example,

```
foreign import capi "pi.h value pi" c_pi :: CDouble
```

will work regardless of whether `pi` is defined as

```
const double pi = 3.14;
```

or with

```
#define pi 3.14
```

In order to tell GHC the C type that a Haskell type corresponds to when it is used with the CAPI, a `CTYPE` pragma can be used on the type definition. The header which defines the type can optionally also be specified. The syntax looks like:

```
data    {-# CTYPE "unistd.h" "useconds_t" #-} T = ...
newtype {-# CTYPE            "useconds_t" #-} T = ...
```

In case foreign declarations contain `const`-qualified pointer return type, `ConstPtr` from Foreign.C.ConstPtr may be used to encode this, e.g.

```
foreign import capi "header.h f" f :: CInt -> ConstPtr CInt
```

which corresponds to

```
const *int f(int);
```

6.17.2.7 `hs_thread_done()`

```
void hs_thread_done(void);
```

GHC allocates a small amount of thread-local memory when a thread calls a Haskell function via a `foreign export`. This memory is not normally freed until `hs_exit()`; the memory is cached so that subsequent calls into Haskell are fast. However, if your application is long-running and repeatedly creates new threads that call into Haskell, you probably want to arrange that this memory is freed in those threads that have finished calling Haskell functions. To do this, call `hs_thread_done()` from the thread whose memory you want to free.

Calling `hs_thread_done()` is entirely optional. You can call it as often or as little as you like. It is safe to call it from a thread that has never called any Haskell functions, or one that never will. If you forget to call it, the worst that can happen is that some memory remains allocated until `hs_exit()` is called. If you call it too often, the worst that can happen is that the next call to a Haskell function incurs some extra overhead.

6.17.2.8 Freeing many stable pointers efficiently

The standard function `hs_free_stable_ptr` locks the stable pointer table, frees the given stable pointer, and then unlocks the stable pointer table again. When freeing many stable pointers at once, it is usually more efficient to lock and unlock the table only once.

```
extern void hs_lock_stable_ptr_table (void);

extern void hs_unlock_stable_ptr_table (void);

extern void hs_free_stable_ptr_unsafe (HsStablePtr sp);
```

`hs_free_stable_ptr_unsafe` must be used *only* when the table has been locked using `hs_lock_stable_ptr_table`. It must be unlocked afterwards using `hs_unlock_stable_ptr_table`. The Haskell garbage collector cannot run while the table is locked, so it should be unlocked promptly. The following operations are forbidden while the stable pointer table is locked:

- Calling any Haskell function, whether or not that function manipulates stable pointers.
- Calling any FFI function that deals with the stable pointer table except for arbitrarily many calls to `hs_free_stable_ptr_unsafe` and the final call to `hs_unlock_stable_ptr_table`.
- Calling `hs_free_fun_ptr`.

Note: GHC versions before 8.8 defined undocumented functions `hs_lock_stable_tables` and `hs_unlock_stable_tables` instead of `hs_lock_stable_ptr_table` and `hs_unlock_stable_ptr_table`. Those names are now deprecated.

6.17.3 Using the FFI with GHC

The following sections also give some hints and tips on the use of the foreign function interface in GHC.

6.17.3.1 Using `foreign export` and `foreign import ccall "wrapper"` with GHC

When GHC compiles a module (say `M.hs`) which uses `foreign export` or `foreign import "wrapper"`, it generates a `M_stub.h` for use by C programs.

For a plain `foreign export`, the file `M_stub.h` contains a C prototype for the foreign exported function. For example, if we compile the following module:

```
module Foo where

foreign export ccall foo :: Int -> IO Int

foo :: Int -> IO Int
foo n = return (length (f n))

f :: Int -> [Int]
f 0 = []
f n = n:(f (n-1))
```

Then `Foo_stub.h` will contain something like this:

```
#include "HsFFI.h"
extern HsInt foo(HsInt a0);
```

To invoke `foo()` from C, just `#include "Foo_stub.h"` and call `foo()`.

The `Foo_stub.h` file can be redirected using the `-stubdir` option; see *Redirecting the compilation output(s)* (page 202).

Using your own `main()`

Normally, GHC's runtime system provides a `main()`, which arranges to invoke `Main.main` in the Haskell program. However, you might want to link some Haskell code into a program which has a main function written in another language, say C. In order to do this, you have to initialize the Haskell runtime system explicitly.

Let's take the example from above, and invoke it from a standalone C program. Here's the C code:

```
#include <stdio.h>
#include "HsFFI.h"

#if defined(__GLASGOW_HASKELL__)
#include "Foo_stub.h"
#endif

int main(int argc, char *argv[])
{
  int i;

  hs_init(&argc, &argv);

  for (i = 0; i < 5; i++) {
    printf("%d\n", foo(2500));
  }

  hs_exit();
  return 0;
}
```

We've surrounded the GHC-specific bits with `#if defined(__GLASGOW_HASKELL__)`; the rest of the code should be portable across Haskell implementations that support the FFI standard.

The call to `hs_init()` initializes GHC's runtime system. Do NOT try to invoke any Haskell functions before calling `hs_init()`: bad things will undoubtedly happen.

We pass references to `argc` and `argv` to `hs_init()` so that it can separate out any arguments for the RTS (i.e. those arguments between `+RTS...-RTS`).

After we've finished invoking our Haskell functions, we can call `hs_exit()`, which terminates the RTS.

There can be multiple calls to `hs_init()`, but each one should be matched by one (and only one) call to `hs_exit()`. The outermost `hs_exit()` will actually de-initialise the system. Note that currently GHC's runtime cannot reliably re-initialise after this has happened; see *The Foreign Function Interface* (page 738).

Note: When linking the final program, it is normally easiest to do the link using GHC, although this isn't essential. If you do use GHC, then don't forget the flag *-no-hs-main* (page 247), otherwise GHC will try to link to the `Main` Haskell module.

Note: On Windows hs_init treats argv as UTF8-encoded. Passing other encodings might lead to unexpected results. Passing NULL as argv is valid but can lead to <unknown> showing up in error messages instead of the name of the executable.

To use +RTS flags with `hs_init()`, we have to modify the example slightly. By default, GHC's RTS will only accept "safe" +RTS flags (see *Options affecting linking* (page 245)), and the *-rtsopts[=⟨none|some|all|ignore|ignoreAll⟩]* (page 248) link-time flag overrides this. However, *-rtsopts[=⟨none|some|all|ignore|ignoreAll⟩]* (page 248) has no effect when *-no-hs-main* (page 247) is in use (and the same goes for *-with-rtsopts=⟨opts⟩* (page 249)). To set these options we have to call a GHC-specific API instead of `hs_init()`:

```
#include <stdio.h>
#include "HsFFI.h"

#if defined(__GLASGOW_HASKELL__)
#include "Foo_stub.h"
#include "Rts.h"
#endif

int main(int argc, char *argv[])
{
  int i;

#if __GLASGOW_HASKELL__ >= 703
  {
      RtsConfig conf = defaultRtsConfig;
      conf.rts_opts_enabled = RtsOptsAll;
      hs_init_ghc(&argc, &argv, conf);
  }
#else
  hs_init(&argc, &argv);
#endif

  for (i = 0; i < 5; i++) {
    printf("%d\n", foo(2500));
  }

  hs_exit();
  return 0;
}
```

Note two changes: we included `Rts.h`, which defines the GHC-specific external RTS interface, and we called `hs_init_ghc()` instead of `hs_init()`, passing an argument of type `RtsConfig`. `RtsConfig` is a struct with various fields that affect the behaviour of the runtime system. Its definition is:

```
typedef struct {
    RtsOptsEnabledEnum rts_opts_enabled;
    const char *rts_opts;
} RtsConfig;
```

(continues on next page)

(continued from previous page)

```
extern const RtsConfig defaultRtsConfig;

typedef enum {
    RtsOptsNone,         // +RTS causes an error
    RtsOptsSafeOnly,     // safe RTS options allowed; others cause an error
    RtsOptsAll           // all RTS options allowed
  } RtsOptsEnabledEnum;
```

There is a default value `defaultRtsConfig` that should be used to initialise variables of type `RtsConfig`. More fields will undoubtedly be added to `RtsConfig` in the future, so in order to keep your code forwards-compatible it is best to initialise with `defaultRtsConfig` and then modify the required fields, as in the code sample above.

Making a Haskell library that can be called from foreign code

The scenario here is much like in *Using your own main()* (page 569), except that the aim is not to link a complete program, but to make a library from Haskell code that can be deployed in the same way that you would deploy a library of C code.

The main requirement here is that the runtime needs to be initialized before any Haskell code can be called, so your library should provide initialisation and deinitialisation entry points, implemented in C or C++. For example:

```
#include <stdlib.h>
#include "HsFFI.h"

HsBool mylib_init(void){
  int argc = 3;
  char *argv[] = { "mylib", "+RTS", "-A32m", NULL };
  char **pargv = argv;

  // Initialize Haskell runtime
  hs_init(&argc, &pargv);

  // do any other initialization here and
  // return false if there was a problem
  return HS_BOOL_TRUE;
}

void mylib_end(void){
  hs_exit();
}
```

The initialisation routine, `mylib_init`, calls `hs_init()` as normal to initialise the Haskell runtime, and the corresponding deinitialisation function `mylib_end()` calls `hs_exit()` to shut down the runtime.

6.17.3.2 Using header files

C functions are normally declared using prototypes in a C header file. Earlier versions of GHC (6.8.3 and earlier) `#include`d the header file in the C source file generated from the Haskell code, and the C compiler could therefore check that the C function being called via the FFI was being called at the right type.

GHC no longer includes external header files when compiling via C, so this checking is not performed. The change was made for compatibility with the *native code generator* (page 236) (*-fasm* (page 243)) and to comply strictly with the FFI specification, which requires that FFI calls are not subject to macro expansion and other CPP conversions that may be applied when using C header files. This approach also simplifies the inlining of foreign calls across module and package boundaries: there's no need for the header file to be available when compiling an inlined version of a foreign call, so the compiler is free to inline foreign calls in any context.

The `-#include` option is now deprecated, and the `include-files` field in a Cabal package specification is ignored.

6.17.3.3 Memory Allocation

The FFI libraries provide several ways to allocate memory for use with the FFI, and it isn't always clear which way is the best. This decision may be affected by how efficient a particular kind of allocation is on a given compiler/platform, so this section aims to shed some light on how the different kinds of allocation perform with GHC.

alloca Useful for short-term allocation when the allocation is intended to scope over a given `IO` computation. This kind of allocation is commonly used when marshalling data to and from FFI functions.

In GHC, `alloca` is implemented using `MutableByteArray#`, so allocation and deallocation are fast: much faster than C's `malloc/free`, but not quite as fast as stack allocation in C. Use `alloca` whenever you can.

mallocForeignPtr Useful for longer-term allocation which requires garbage collection. If you intend to store the pointer to the memory in a foreign data structure, then `mallocForeignPtr` is *not* a good choice, however.

In GHC, `mallocForeignPtr` is also implemented using `MutableByteArray#`. Although the memory is pointed to by a `ForeignPtr`, there are no actual finalizers involved (unless you add one with `addForeignPtrFinalizer`), and the deallocation is done using GC, so `mallocForeignPtr` is normally very cheap.

malloc/free If all else fails, then you need to resort to `Foreign.malloc` and `Foreign.free`. These are just wrappers around the C functions of the same name, and their efficiency will depend ultimately on the implementations of these functions in your platform's C library. We usually find `malloc` and `free` to be significantly slower than the other forms of allocation above.

Foreign.Marshal.Pool Pools can be a more convenient way to structure your memory allocation than using one of the other forms of allocation. They are backed by the RTS internal arena instead of `malloc/free`.

6.17.3.4 Multi-threading and the FFI

In order to use the FFI in a multi-threaded setting, you must use the *-threaded* (page 248) option (see *Options affecting linking* (page 245)).

Foreign imports and multi-threading

When you call a `foreign imported` function that is annotated as `safe` (the default) in a single-threaded runtime (the program was linked without using *-threaded* (page 248)), then other Haskell threads will be blocked until the call returns.

In the multi-threaded runtime (the program was linked using *-threaded* (page 248)), `foreign imported` functions run concurrently (both `safe` and `unsafe`), but a similar effect can happen when you call an `unsafe` function, and a global garbage collection is triggered in another thread. In this situation, the garbage collector cannot proceed, and this can lead to performance issues that often appear under high load, as other threads are more active and thus more prone to trigger global garbage collection.

This means that if you need to make a foreign call to a function that takes a long time or potentially blocks, then you should mark it `safe` and use *-threaded* (page 248). Some library functions make such calls internally; their documentation should indicate when this is the case.

On the other hand, a foreign call to a function that is guaranteed to take a short time, and does not call back into Haskell can be marked `unsafe`. This works both for the single-threaded and the multi-threaded runtime. When considering what "a short time" is, a foreign function that does comparable work to what Haskell code does between each heap allocation (not very much), is a good candidate.

Outside these two clear cases for `safe` and `unsafe` foreign functions, there is a trade-off between whole-program throughput and efficiency of the individual foreign function call.

If you are making foreign calls from multiple Haskell threads and using *-threaded* (page 248), make sure that the foreign code you are calling is thread-safe. In particularly, some GUI libraries are not thread-safe and require that the caller only invokes GUI methods from a single thread. If this is the case, you may need to restrict your GUI operations to a single Haskell thread, and possibly also use a bound thread (see *The relationship between Haskell threads and OS threads* (page 574)).

Note that foreign calls made by different Haskell threads may execute in *parallel*, even when the `+RTS -N` flag is not being used (*RTS options for SMP parallelism* (page 132)). The *-N ⟨x⟩* (page 132) flag controls parallel execution of Haskell threads, but there may be an arbitrary number of foreign calls in progress at any one time, regardless of the `+RTS -N` value.

If a call is annotated as `interruptible` and the program was multithreaded, the call may be interrupted in the event that the Haskell thread receives an exception. The mechanism by which the interrupt occurs is platform dependent, but is intended to cause blocking system calls to return immediately with an interrupted error code. The underlying operating system thread is not to be destroyed. See *Interruptible foreign calls* (page 565) for more details.

The relationship between Haskell threads and OS threads

Normally there is no fixed relationship between Haskell threads and OS threads. This means that when you make a foreign call, that call may take place in an unspecified OS thread. Furthermore, there is no guarantee that multiple calls made by one Haskell thread will be made by the same OS thread.

This usually isn't a problem, and it allows the GHC runtime system to make efficient use of OS thread resources. However, there are cases where it is useful to have more control over which OS thread is used, for example when calling foreign code that makes use of thread-local state. For cases like this, we provide *bound threads*, which are Haskell threads tied to a particular OS thread. For information on bound threads, see the documentation for the Control.Concurrent module.

Foreign exports and multi-threading

When the program is linked with *`-threaded`* (page 248), then you may invoke `foreign exported` functions from multiple OS threads concurrently. The runtime system must be initialised as usual by calling `hs_init()`, and this call must complete before invoking any `foreign exported` functions.

On the use of `hs_exit()`

`hs_exit()` normally causes the termination of any running Haskell threads in the system, and when `hs_exit()` returns, there will be no more Haskell threads running. The runtime will then shut down the system in an orderly way, generating profiling output and statistics if necessary, and freeing all the memory it owns.

It isn't always possible to terminate a Haskell thread forcibly: for example, the thread might be currently executing a foreign call, and we have no way to force the foreign call to complete. What's more, the runtime must assume that in the worst case the Haskell code and runtime are about to be removed from memory (e.g. if this is a *Windows DLL* (page 706), `hs_exit()` is normally called before unloading the DLL). So `hs_exit()` *must* wait until all outstanding foreign calls return before it can return itself.

The upshot of this is that if you have Haskell threads that are blocked in foreign calls, then `hs_exit()` may hang (or possibly busy-wait) until the calls return. Therefore it's a good idea to make sure you don't have any such threads in the system when calling `hs_exit()`. This includes any threads doing I/O, because I/O may (or may not, depending on the type of I/O and the platform) be implemented using blocking foreign calls.

The GHC runtime treats program exit as a special case, to avoid the need to wait for blocked threads when a standalone executable exits. Since the program and all its threads are about to terminate at the same time that the code is removed from memory, it isn't necessary to ensure that the threads have exited first. If you want this fast and loose version of `hs_exit()`, you can call:

```
void hs_exit_nowait(void);
```

instead. This is particularly useful if you have foreign libraries that need to call `hs_exit()` at program exit (perhaps via a C++ destructor): in this case you should use `hs_exit_nowait()`, because the thread that called `exit()` and is running C++ destructors is in a foreign call from Haskell that will never return, so `hs_exit()` would deadlock.

Waking up Haskell threads from C

Sometimes we want to be able to wake up a Haskell thread from some C code. For example, when using a callback-based C API, we register a C callback and then we need to wait for the callback to run.

One way to do this is to create a `foreign export` that will do whatever needs to be done to wake up the Haskell thread - perhaps `putMVar` - and then call this from our C callback. There are a couple of problems with this:

1. Calling a foreign export has a lot of overhead: it creates a complete new Haskell thread, for example.
2. The call may block for a long time if a GC is in progress. We can't use this method if the C API we're calling doesn't allow blocking in the callback.

For these reasons GHC provides an external API to `tryPutMVar`, `hs_try_putmvar`, which you can use to cheaply and asynchronously wake up a Haskell thread from C/C++.

```
void hs_try_putmvar (int capability, HsStablePtr sp);
```

The C call `hs_try_putmvar(cap, mvar)` is equivalent to the Haskell call `tryPutMVar mvar ()`, except that it is

- non-blocking: takes a bounded, short, amount of time
- asynchronous: the actual putMVar may be performed after the call returns (for example, if the RTS is currently garbage collecting). That's why `hs_try_putmvar()` doesn't return a result to say whether the put succeeded. It is your responsibility to ensure that the `MVar` is empty; if it is full, `hs_try_putmvar()` will have no effect.

Example. Suppose we have a C/C++ function to call that will return and then invoke a callback at some point in the future, passing us some data. We want to wait in Haskell for the callback to be called, and retrieve the data. We can do it like this:

```
import GHC.Conc (newStablePtrPrimMVar, PrimMVar)

makeExternalCall = mask_ $ do
  mvar <- newEmptyMVar
  sp <- newStablePtrPrimMVar mvar
  fp <- mallocForeignPtr
  withForeignPtr fp $ \presult -> do
    cap <- threadCapability =<< myThreadId
    scheduleCallback sp cap presult
    takeMVar mvar `onException`
      forkIO (do takeMVar mvar; touchForeignPtr fp)
    peek presult

foreign import ccall "scheduleCallback"
    scheduleCallback :: StablePtr PrimMVar
                     -> Int
                     -> Ptr Result
                     -> IO ()
```

And inside `scheduleCallback`, we create a callback that will in due course store the result data in the `Ptr Result`, and then call `hs_try_putmvar()`.

There are a few things to note here.

- There's a special function to create the `StablePtr`: `newStablePtrPrimMVar`, because the RTS needs a `StablePtr` to the primitive `MVar#` object, and we can't create that directly. Do *not* just use `newStablePtr` on the `MVar`: your program will crash.
- The `StablePtr` is freed by `hs_try_putmvar()`. This is because it would otherwise be difficult to arrange to free the `StablePtr` reliably: we can't free it in Haskell, because if the `takeMVar` is interrupted by an asynchronous exception, then the callback will fire at a later time. We can't free it in C, because we don't know when to free it (not when `hs_try_putmvar()` returns, because that is an async call that uses the `StablePtr` at some time in the future).
- The `mask_` is to avoid asynchronous exceptions before the `scheduleCallback` call, which would leak the `StablePtr`.
- We find out the current capability number and pass it to C. This is passed back to `hs_try_putmvar`, and helps the RTS to know which capability it should try to perform the `tryPutMVar` on. If you don't care, you can pass `-1` for the capability to `hs_try_putmvar`, and it will pick an arbitrary one.

  Picking the right capability will help avoid unnecessary context switches. Ideally you should pass the capability that the thread that will be woken up last ran on, which you can find by calling `threadCapability` in Haskell.
- If you want to also pass some data back from the C callback to Haskell, this is best done by first allocating some memory in Haskell to receive the data, and passing the address to C, as we did in the above example.
- `takeMVar` can be interrupted by an asynchronous exception. If this happens, the callback in C will still run at some point in the future, will still write the result, and will still call `hs_try_putmvar()`. Therefore we have to arrange that the memory for the result stays alive until the callback has run, so if an exception is thrown during `takeMVar` we fork another thread to wait for the callback and hold the memory alive using `touchForeignPtr`.

For a fully working example, see `testsuite/tests/concurrent/should_run/hs_try_putmvar001.hs` in the GHC source tree.

6.17.3.5 Floating point and the FFI

The standard C99 `fenv.h` header provides operations for inspecting and modifying the state of the floating point unit. In particular, the rounding mode used by floating point operations can be changed, and the exception flags can be tested.

In Haskell, floating-point operations have pure types, and the evaluation order is unspecified. So strictly speaking, since the `fenv.h` functions let you change the results of, or observe the effects of floating point operations, use of `fenv.h` renders the behaviour of floating-point operations anywhere in the program undefined.

Having said that, we *can* document exactly what GHC does with respect to the floating point state, so that if you really need to use `fenv.h` then you can do so with full knowledge of the pitfalls:

- GHC completely ignores the floating-point environment, the runtime neither modifies nor reads it.
- The floating-point environment is not saved over a normal thread context-switch. So if you modify the floating-point state in one thread, those changes may be visible in other threads. Furthermore, testing the exception state is not reliable, because a context switch may change it. If you need to modify or test the floating point state and use threads, then you must use bound threads (`Control.Concurrent.forkOS`), because a

bound thread has its own OS thread, and OS threads do save and restore the floating-point state.

- It is safe to modify the floating-point unit state temporarily during a foreign call, because foreign calls are never pre-empted by GHC.

6.17.3.6 Pinned Byte Arrays

A pinned byte array is one that the garbage collector is not allowed to move. Consequently, it has a stable address that can be safely requested with `byteArrayContents#`. Not that being pinned doesn't prevent the byteArray from being gc'ed in the same fashion a regular byte array would be. There are a handful of primitive functions in GHC.Exts used to enforce or check for pinnedness: `isByteArrayPinned#`, `isMutableByteArrayPinned#`, and `newPinnedByteArray#`. A byte array can be pinned as a result of three possible causes:

1. It was allocated by `newPinnedByteArray#`.
2. It is large. Currently, GHC defines large object to be one that is at least as large as 80% of a 4KB block (i.e. at least 3277 bytes).
3. It has been copied into a compact region. The documentation for `ghc-compact` and `compact` describes this process.

6.18 Safe Haskell

Safe Haskell is an extension to the Haskell language that is implemented in GHC as of version 7.2. It allows for unsafe code to be securely included in a trusted code base by restricting the features of GHC Haskell the code is allowed to use. Put simply, it makes the types of programs trustable.

While a primary use case of Safe Haskell is running untrusted code, Safe Haskell doesn't provide this directly. Instead, Safe Haskell provides strict type safety. Without Safe Haskell, GHC allows many exceptions to the type system which can subvert any abstractions. By providing strict type safety, Safe Haskell enables developers to build their own library level sandbox mechanisms to run untrusted code.

While Safe Haskell is an extension, it actually runs in the background for every compilation with GHC. It does this to track the type violations of modules to infer their safety, even when they aren't explicitly using Safe Haskell. Please refer to section *Safe Haskell Inference* (page 586) for more details of this.

The design of Safe Haskell covers the following aspects:

- A *safe language* (page 580) dialect of Haskell that provides stricter guarantees about the code. It allows types and module boundaries to be trusted.
- A *safe import* extension that specifies that the module being imported must be trusted.
- A definition of *trust* (or safety) and how it operates, along with ways of defining and changing the trust of modules and packages.

Safe Haskell, however, *does not offer* compilation safety. During compilation time it is possible for arbitrary processes to be launched, using for example the *custom pre-processor* (page 243) flag. This can be manipulated to either compromise a user's system at compilation time, or to modify the source code just before compilation to try to alter Safe Haskell flags. This is discussed further in section *Safe Compilation* (page 588).

6.18.1 Uses of Safe Haskell

Safe Haskell has been designed with two use cases in mind:

- Enforcing strict type safety at compile time
- Compiling and executing untrusted code

6.18.1.1 Strict type-safety (good style)

Haskell offers a powerful type system and separation of pure and effectual functions through the `IO` monad. However, there are several loop holes in the type system, the most obvious being the `unsafePerformIO :: IO a -> a` function. The safe language dialect of Safe Haskell disallows the use of such functions. This can be useful restriction as it makes Haskell code easier to analyse and reason about. It also codifies the existing culture in the Haskell community of trying to avoid unsafe functions unless absolutely necessary. As such, using the safe language (through the `-XSafe` flag) can be thought of as a way of enforcing good style, similar to the function of `-Wall`.

6.18.1.2 Building secure systems (restricted IO Monads)

Systems such as information flow control security, capability based security systems and DSLs for working with encrypted data.. etc can be built in the Haskell language as a library. However they require guarantees about the properties of Haskell that aren't true in general due to the presence of functions like `unsafePerformIO`. Safe Haskell gives users enough guarantees about the type system to allow them to build such secure systems.

As an example, let's define an interface for a plugin system where the plugin authors are untrusted, possibly malicious third-parties. We do this by restricting the plugin interface to pure functions or to a restricted `IO` monad that we have defined. The restricted `IO` monad will only allow a safe subset of `IO` actions to be executed. We define the plugin interface so that it requires the plugin module, `Danger`, to export a single computation, `Danger.runMe`, of type `RIO ()`, where `RIO` is a monad defined as follows:

```haskell
-- While we use `Safe', the `Trustworthy' pragma would also be
-- fine. We simply want to ensure that:
-- 1) The module exports an interface that untrusted code can't
--    abuse.
-- 2) Untrusted code can import this module.
--
{-# LANGUAGE Safe #-}

module RIO (RIO(), runRIO, rioReadFile, rioWriteFile) where

-- Notice that symbol UnsafeRIO is not exported from this module!
newtype RIO a = UnsafeRIO { runRIO :: IO a }

instance Functor RIO where
    fmap f (UnsafeRIO m) = UnsafeRIO (fmap f m)

instance Applicative RIO where
    pure = UnsafeRIO . pure
    (UnsafeRIO f) <*> (UnsafeRIO m) = UnsafeRIO (f <*> m)

instance Monad RIO where
```

(continues on next page)

(continued from previous page)

```
    (UnsafeRIO m) >>= k = UnsafeRIO $ m >>= runRIO . k

-- Returns True iff access is allowed to file name
pathOK :: FilePath -> IO Bool
pathOK file = {- Implement some policy based on file name -}

rioReadFile :: FilePath -> RIO String
rioReadFile file = UnsafeRIO $ do
    ok <- pathOK file
    if ok then readFile file else return ""

rioWriteFile :: FilePath -> String -> RIO ()
rioWriteFile file contents = UnsafeRIO $ do
    ok <- pathOK file
    if ok then writeFile file contents else return ()
```

We then compile the `Danger` plugin using the new Safe Haskell `-XSafe` flag:

```
{-# LANGUAGE Safe #-}
module Danger ( runMe ) where

runMe :: RIO ()
runMe = ...
```

Before going into the Safe Haskell details, let's point out some of the reasons this security mechanism would fail without Safe Haskell:

- The design attempts to restrict the operations that `Danger` can perform by using types, specifically the `RIO` type wrapper around `IO` . The author of `Danger` can subvert this though by simply writing arbitrary `IO` actions and using `unsafePerformIO :: IO a -> a` to execute them as pure functions.
- The design also relies on `Danger` not being able to access the `UnsafeRIO` constructor. Unfortunately Template Haskell can be used to subvert module boundaries and so could be used to gain access to this constructor.
- There is no way to place restrictions on the modules that `Danger` can import. This gives the author of `Danger` a very large attack surface, essentially any package currently installed on the system. Should any of these packages have a vulnerability, then the `Danger` module can exploit it.

Safe Haskell prevents all these attacks. This is done by compiling the RIO module with the *`Safe`* (page 586) or *`Trustworthy`* (page 586) flag and compiling `Danger` with the *`Safe`* (page 586) flag. We explain each below.

The use of *`Safe`* (page 586) to compile `Danger` restricts the features of Haskell that can be used to a *safe subset* (page 580). This includes disallowing `unsafePerformIO`, Template Haskell, pure FFI functions, RULES and restricting the operation of Overlapping Instances. The *`Safe`* (page 586) flag also restricts the modules can be imported by `Danger` to only those that are considered trusted. Trusted modules are those compiled with *`Safe`* (page 586), where GHC provides a mechanical guarantee that the code is safe. Or those modules compiled with *`Trustworthy`* (page 586), where the module author claims that the module is Safe.

This is why the RIO module is compiled with *`Safe`* (page 586) or *`Trustworthy`* (page 586), to allow the `Danger` module to import it. The *`Trustworthy`* (page 586) flag doesn't place any restrictions on the module like *`Safe`* (page 586) does (expect to restrict overlapping instances to *safe overlapping instances* (page 581)). Instead the module author claims that while code may use unsafe features internally, it only exposes an API that can used in a safe manner.

However, the unrestricted use of *Trustworthy* (page 586) is a problem as an arbitrary module can use it to mark themselves as trusted, yet *Trustworthy* (page 586) doesn't offer any guarantees about the module, unlike *Safe* (page 586). To control the use of trustworthy modules it is recommended to use the *-fpackage-trust* (page 587) flag. This flag adds an extra requirement to the trust check for trustworthy modules. It requires that for a trustworthy modules to be considered trusted, and allowed to be used in *Safe* (page 586) compiled code, the client C compiling the code must tell GHC that they trust the package the trustworthy module resides in. This is essentially a way of for C to say, while this package contains trustworthy modules that can be used by untrusted modules compiled with *Safe* (page 586), I trust the author(s) of this package and trust the modules only expose a safe API. The trust of a package can be changed at any time, so if a vulnerability found in a package, C can declare that package untrusted so that any future compilation against that package would fail. For a more detailed overview of this mechanism see *Trust and Safe Haskell Modes* (page 583).

In the example, `Danger` can import module `RIO` because `RIO` is compiled with *Safe* (page 586). Thus, `Danger` can make use of the `rioReadFile` and `rioWriteFile` functions to access permitted file names. The main application then imports both `RIO` and `Danger`. To run the plugin, it calls `RIO.runRIO Danger.runMe` within the `IO` monad. The application is safe in the knowledge that the only `IO` to ensue will be to files whose paths were approved by the `pathOK` test.

The Safe Haskell checks can be disabled for a module by passing the *-fno-safe-haskell* (page 587) flag. This is useful in particular when compiling with source plugins as running a plugin marks the module as unsafe and can then cause downstream modules to fail the safety checks.

6.18.2 Safe Language

The Safe Haskell *safe language* (enabled by `-XSafe`) guarantees the following properties:

- *Referential transparency* — The types can be trusted. Any pure function, is guaranteed to be pure. Evaluating them is deterministic and won't cause any side effects. Functions in the `IO` monad are still allowed and behave as usual. So, for example, the `unsafePerformIO :: IO a -> a` function is disallowed in the safe language to enforce this property.
- *Module boundary control* — Only symbols that are publicly available through other module export lists can be accessed in the safe language. Values using data constructors not exported by the defining module, cannot be examined or created. As such, if a module `M` establishes some invariants through careful use of its export list, then code written in the safe language that imports `M` is guaranteed to respect those invariants.
- *Semantic consistency* — For any module that imports a module written in the safe language, expressions that compile both with and without the safe import have the same meaning in both cases. That is, importing a module written in the safe language cannot change the meaning of existing code that isn't dependent on that module. So, for example, there are some restrictions placed on the use of *OverlappingInstances* (page 486), as these can violate this property.
- *Strict subset* — The safe language is strictly a subset of Haskell as implemented by GHC. Any expression that compiles in the safe language has the same meaning as it does when compiled in normal Haskell.

These four properties guarantee that in the safe language you can trust the types, can trust that module export lists are respected, and can trust that code that successfully compiles has the same meaning as it normally would.

To achieve these properties, in the safe language dialect we disable completely the following features:

- *TemplateHaskell* (page 528) — Can be used to gain access to constructors and abstract data types that weren't exported by a module, subverting module boundaries.

Furthermore, we restrict the following features:

- *ForeignFunctionInterface* (page 561) — Foreign import declarations that import a function with a non-IO type are disallowed.
- RULES — Rewrite rules defined in a module M compiled with *Safe* (page 586) are dropped. Rules defined in Trustworthy modules that M imports are still valid and will fire as usual.
- *OverlappingInstances* (page 486) — There is no restriction on the creation of overlapping instances, but we do restrict their use at a particular call site. This is a detailed restriction, please refer to *Safe Overlapping Instances* (page 581) for details.
- *GeneralisedNewtypeDeriving* (page 447) and *DerivingVia* (page 457) — GND and DerivingVia are not allowed in the safe language. This is due to the ability of them to violate module boundaries when module authors forget to put nominal role annotations on their types as appropriate. For this reason, the Data.Coerce module is also considered unsafe. We are hoping to find a better solution here in the future.
- GHC.Generics — Hand crafted instances of the Generic type class are not allowed in Safe Haskell. Such instances aren't strictly unsafe, but there is an important invariant that a Generic instance should adhere to the structure of the data type for which the instance is defined, and allowing manually implemented Generic instances would break that invariant. Derived instances (through the *DeriveGeneric* (page 598) extension) are still allowed. Note that the only allowed *deriving strategy* (page 455) for deriving Generic under Safe Haskell is stock, as another strategy (e.g., anyclass) would produce an instance that violates the invariant.

  Refer to the *generic programming* (page 597) section for more details.

6.18.2.1 Safe Overlapping Instances

Due to the semantic consistency guarantee of Safe Haskell, we must restrict the function of overlapping instances. We don't restrict their ability to be defined, as this is a global property and not something we can determine by looking at a single module. Instead, when a module calls a function belonging to a type-class, we check that the instance resolution done is considered 'safe'. This check is enforced for modules compiled with both -XSafe and -XTrustworthy.

More specifically, consider the following modules:

```haskell
{-# LANGUAGE Safe #-}
module Class (TC(..)) where
  class TC a where { op :: a -> String }

{-# LANGUAGE Safe #-}
module Dangerous (TC(..)) where
  import Class

  instance
    {-# OVERLAPS #-}
```

(continues on next page)

(continued from previous page)

```
    TC [Int] where { op _ = "[Int]" }

{-# LANGUAGE Safe #-}
module TCB_Runner where
  import Class
  import Dangerous

  instance
    TC [a] where { op _ = "[a]" }

  f :: String
  f = op ([1,2,3,4] :: [Int])
```

Both module `Class` and module `Dangerous` will compile under *`Safe`* (page 586) without issue. However, in module `TCB_Runner`, we must check if the call to `op` in function `f` is safe.

What does it mean to be Safe? That importing a module compiled with *`Safe`* (page 586) shouldn't change the meaning of code that compiles fine without importing the module. This is the Safe Haskell property known as *semantic consistency*.

In our situation, module `TCB_Runner` compiles fine without importing module `Dangerous`. So when deciding which instance to use for the call to `op`, if we determine the instance `TC [Int]` from module Dangerous is the most specific, this is unsafe. This prevents code written by third-parties we don't trust (which is compiled using `-XSafe` in Safe Haskell) from changing the behaviour of our existing code.

Specifically, we apply the following rule to determine if a type-class method call is *unsafe* when overlapping instances are involved:

- Most specific instance, `Ix`, defined in an `-XSafe` compiled module.
- `Ix` is an orphan instance or a multi-parameter-type-class.
- At least one overlapped instance, `Iy`, is both:
  - From a different module than `Ix`
  - `Iy` is not marked `OVERLAPPABLE`

This is a slightly involved heuristic, but captures the situation of an imported module `N` changing the behaviour of existing code. For example, if the second condition isn't violated, then the module author `M` must depend either on a type-class or type defined in `N`.

When a particular type-class method call is considered unsafe due to overlapping instances, and the module being compiled is using *`Safe`* (page 586) or *`Trustworthy`* (page 586), then compilation will fail. For *`Unsafe`* (page 587), no restriction is applied, and for modules using safe inference, they will be inferred unsafe.

6.18.3 Safe Imports

Safe Haskell enables a small extension to the usual import syntax of Haskell, adding a `safe` keyword:

```
impdecl -> import [safe] [qualified] modid [as modid] [impspec]
```

When used, the module being imported with the safe keyword must be a trusted module, otherwise a compilation error will occur. The safe import extension is enabled by either of the `-XSafe` , `-XTrustworthy` , or `-XUnsafe` flags. When the `-XSafe` flag is used, the `safe` keyword is allowed but meaningless, as every import is treated as a safe import.

6.18.4 Trust and Safe Haskell Modes

Safe Haskell introduces the following three language flags:

- *`Safe`* (page 586) — Enables the safe language dialect, asking GHC to guarantee trust. The safe language dialect requires that all imports be trusted or a compilation error will occur. Safe Haskell will also infer this safety type for modules automatically when possible. Please refer to section *Safe Haskell Inference* (page 586) for more details of this.
- *`Trustworthy`* (page 586) — Means that while this module may invoke unsafe functions internally, the module's author claims that it exports an API that can't be used in an unsafe way. This doesn't enable the safe language. It does however restrict the resolution of overlapping instances to only allow *safe overlapping instances* (page 581). The trust guarantee is provided by the module author, not GHC. An import statement with the `safe` keyword results in a compilation error if the imported module is not trusted. An import statement without the keyword behaves as usual and can import any module whether trusted or not.
- *`Unsafe`* (page 587) — Marks the module being compiled as unsafe so that modules compiled using *`Safe`* (page 586) can't import it. You may want to explicitly mark a module unsafe when it exports internal constructors that can be used to violate invariants.

While these are flags, they also correspond to Safe Haskell module types that a module can have. You can think of using these as declaring an explicit contract (or type) that a module must have. If it is invalid, then compilation will fail. GHC will also infer the correct type for Safe Haskell, please refer to section *Safe Haskell Inference* (page 586) for more details.

The procedure to check if a module is trusted or not depends on if the *`-fpackage-trust`* (page 587) flag is present. The check is similar in both cases with the *`-fpackage-trust`* (page 587) flag enabling an extra requirement for trustworthy modules to be regarded as trusted.

6.18.4.1 Trust check (`-fpackage-trust` disabled)

A module `M` in a package `P` is trusted by a client C if and only if:

- Both of these hold:
  - The module was compiled with *`Safe`* (page 586)
  - All of M's direct imports are trusted by C
- *or* all of these hold:
  - The module was compiled with *`Trustworthy`* (page 586)

- All of `M`'s direct *safe imports* are trusted by C

The above definition of trust has an issue. Any module can be compiled with *Trustworthy* (page 586) and it will be trusted. To control this, there is an additional definition of package trust (enabled with the *-fpackage-trust* (page 587) flag). The point of package trust is to require that the client C explicitly say which packages are allowed to contain trustworthy modules. Trustworthy packages are only trusted if they reside in a package trusted by C.

6.18.4.2 Trust check (`-fpackage-trust` enabled)

When the *-fpackage-trust* (page 587) flag is enabled, whether or not a module is trusted depends on if certain packages are trusted. Package trust is determined by the client C invoking GHC (i.e. you).

Specifically, a package *P is trusted* when one of these hold:

- C's package database records that `P` is trusted (and no command-line arguments override this)
- C's command-line flags say to trust `P` regardless of what is recorded in the package database.

In either case, C is the only authority on package trust. It is up to the client to decide which *packages they trust* (page 585).

When the *-fpackage-trust* (page 587) flag is used a *module M from package P is trusted by a client C* if and only if:

- Both of these hold:
  - The module was compiled with *Safe* (page 586)
  - All of `M`'s direct imports are trusted by C
- *or* all of these hold:
  - The module was compiled with *Trustworthy* (page 586)
  - All of `M`'s direct safe imports are trusted by C
  - Package `P` is trusted by C

For the first trust definition the trust guarantee is provided by GHC through the restrictions imposed by the safe language. For the second definition of trust, the guarantee is provided initially by the module author. The client C then establishes that they trust the module author by indicating they trust the package the module resides in. This trust chain is required as GHC provides no guarantee for *Trustworthy* (page 586) compiled modules.

The reason there are two modes of checking trust is that the extra requirement enabled by *-fpackage-trust* (page 587) causes the design of Safe Haskell to be invasive. Packages using Safe Haskell when the flag is enabled may or may not compile depending on the state of trusted packages on a user's machine. This is both fragile, and causes compilation failures for everyone, even if they aren't trying to use any of the guarantees provided by Safe Haskell. Disabling *-fpackage-trust* (page 587) by default and turning it into a flag makes Safe Haskell an opt-in extension rather than an always on feature.

6.18.4.3 Example

```
Package Wuggle:
    {-# LANGUAGE Safe #-}
    module Buggle where
        import Prelude
        f x = ...blah...

Package P:
    {-# LANGUAGE Trustworthy #-}
    module M where
        import System.IO.Unsafe
        import safe Buggle
```

Suppose a client C decides to trust package `P` and package `base`. Then does C trust module `M`? Well `M` is marked *`Trustworthy`* (page 586), so we don't restrict the language. However, we still must check `M`'s imports:

- First, `M` imports `System.IO.Unsafe`. This is an unsafe module, however `M` was compiled with *`Trustworthy`* (page 586) , so `P`'s author takes responsibility for that import. `C` trusts `P`'s author, so this import is fine.
- Second, `M` safe imports `Buggle`. For this import `P`'s author takes no responsibility for the safety, instead asking GHC to check whether `Buggle` is trusted by `C`. Is it?
- `Buggle`, is compiled with `-XSafe`, so the code is machine-checked to be OK, but again under the assumption that all of `Buggle`'s imports are trusted by `C`. We must recursively check all imports!
- Buggle only imports `Prelude`, which is compiled with *`Trustworthy`* (page 586). `Prelude` resides in the `base` package, which `C` trusts, and (we'll assume) all of `Prelude`'s imports are trusted. So `C` trusts `Prelude`, and so `C` also trusts Buggle. (While `Prelude` is typically imported implicitly, it still obeys the same rules outlined here).

Notice that C didn't need to trust package Wuggle; the machine checking is enough. C only needs to trust packages that contain *`Trustworthy`* (page 586) modules.

6.18.4.4 Trustworthy Requirements

Module authors using the *`Trustworthy`* (page 586) language extension for a module `M` should ensure that `M`'s public API (the symbols exposed by its export list) can't be used in an unsafe manner. This mean that symbols exported should respect type safety and referential transparency.

6.18.4.5 Package Trust

Safe Haskell gives packages a new Boolean property, that of trust. Several new options are available at the GHC command-line to specify the trust property of packages:

`-trust` ⟨pkg⟩
: Exposes package ⟨pkg⟩ if it was hidden and considers it a trusted package regardless of the package database.

`-distrust` ⟨pkg⟩
: Exposes package ⟨pkg⟩ if it was hidden and considers it an untrusted package regardless of the package database.

-distrust-all-packages
Considers all packages distrusted unless they are explicitly set to be trusted by subsequent command-line options.

To set a package's trust property in the package database please refer to *Packages* (page 219).

6.18.5 Safe Haskell Inference

In the case where a module is compiled without one of `Safe` (page 586), `Trustworthy` (page 586) or `Unsafe` (page 587) being used, GHC will try to figure out itself if the module can be considered safe. This safety inference will never mark a module as trustworthy, only as either unsafe or as safe. GHC uses a simple method to determine this for a module M: If M would compile without error under the `Safe` (page 586) flag, then M is marked as safe. Otherwise, it is marked as unsafe.

When should you use Safe Haskell inference and when should you use an explicit `Safe` (page 586) flag? The later case should be used when you have a hard requirement that the module be safe. This is most useful for the *Uses of Safe Haskell* (page 578) of Safe Haskell: running untrusted code. Safe inference is meant to be used by ordinary Haskell programmers. Users who probably don't care about Safe Haskell.

Haskell library authors have a choice. Most should just use Safe inference. Assuming you avoid any unsafe features of the language then your modules will be marked safe. Inferred vs. Explicit has the following trade-offs:

- *Inferred* — This works well and adds no dependencies on the Safe Haskell type of any modules in other packages. It does mean that the Safe Haskell type of your own modules could change without warning if a dependency changes. One way to deal with this is through the use of *Safe Haskell warning flags* (page 586) that will warn if GHC infers a Safe Haskell type different from expected.
- *Explicit* — This gives your library a stable Safe Haskell type that others can depend on. However, it will increase the chance of compilation failure when your package dependencies change.

6.18.6 Safe Haskell Flag Summary

In summary, Safe Haskell consists of the following three language flags:

Safe

Since 7.2.1

Restricts the module to the safe language. All of the module's direct imports must be trusted, but the module itself need not reside in a trusted package, because the compiler vouches for its trustworthiness. The "safe" keyword is allowed but meaningless in import statements, as regardless, every import is required to be safe.

- *Module Trusted* — Yes
- *Haskell Language* — Restricted to Safe Language
- *Imported Modules* — All forced to be safe imports, all must be trusted.

Trustworthy

Since 7.2.1

This establishes that the module is trusted, but the guarantee is provided by the module's author. A client of this module then specifies that they trust the module author by specifying they trust the package containing the module. *Trustworthy* (page 586) doesn't restrict the module to the safe language. It does however restrict the resolution of overlapping instances to only allow *safe overlapping instances* (page 581). It also allows the use of the safe import keyword.

- *Module Trusted* — Yes.
- *Module Trusted* (*-fpackage-trust* (page 587) enabled) — Yes but only if the package the module resides in is also trusted.
- *Haskell Language* — Unrestricted, except only safe overlapping instances allowed.
- *Imported Modules* — Under control of module author which ones must be trusted.

Unsafe

Since 7.4.1

Mark a module as unsafe so that it can't be imported by code compiled with *Safe* (page 586). Also enable the Safe Import extension so that a module can require a dependency to be trusted.

- *Module Trusted* — No
- *Haskell Language* — Unrestricted
- *Imported Modules* — Under control of module author which ones must be trusted.

A flag to disable Safe Haskell checks:

-fno-safe-haskell
This flag can be enabled to override any declared safety property of the module (Safe, Unsafe, Trustworthy) so compilation proceeds as if none of these flags were specified. This is particularly useful when compiling using plugins, which usually results in the compiled modules being marked as unsafe.

And one general flag:

-fpackage-trust
When enabled, turn on an extra check for a trustworthy module `M`, requiring the package that `M` resides in be considered trusted, for `M` to be considered trusted.

And five warning flags:

-Wunsafe
Issue a warning if the module being compiled is regarded to be unsafe. Should be used to check the safety type of modules when using safe inference.

-Wsafe
Issue a warning if the module being compiled is regarded to be safe. Should be used to check the safety type of modules when using safe inference. If the module is explicitly marked as safe then no warning will be issued.

-Wtrustworthy-safe
Issue a warning if the module being compiled is marked as -XTrustworthy but it could instead be marked as -XSafe , a more informative bound. Can be used to detect once a Safe Haskell bound can be improved as dependencies are updated.

-Winferred-safe-imports

Since 8.10.1

Default off

The module A below is annotated to be explicitly Safe, but it imports Safe-Inferred module.

```
{-# LANGUAGE Safe #-}
module A where

import B (double)

quad :: Int -> Int
quad = double . double


module B where

double :: Int -> Int
double n = n + n
```

The inferred status is volatile: if an unsafe import is added to the module B, it will cause compilation error of A. When *-Winferred-safe-imports* (page 587) is enabled, the compiler will emit a warning about this.

-Wmissing-safe-haskell-mode

Since 8.10.1

Default off

The compiler will warn when none of *Safe* (page 586), *Trustworthy* (page 586) or *Unsafe* (page 587) is specified.

6.18.7 Safe Compilation

GHC includes a variety of flags that allow arbitrary processes to be run at compilation time. One such example is the *custom pre-processor* (page 243) flag. Another is the ability of Template Haskell to execute Haskell code at compilation time, including IO actions. Safe Haskell *does not address this danger* (although, Template Haskell is a disallowed feature).

Due to this, it is suggested that when compiling untrusted source code that has had no manual inspection done, the following precautions be taken:

- Compile in a sandbox, such as a chroot or similar container technology. Or simply as a user with very reduced system access.
- Compile untrusted code with the -XSafe flag being specified on the command line. This will ensure that modifications to the source being compiled can't disable the use of the Safe Language as the command line flag takes precedence over a source level pragma.
- Ensure that all untrusted code is imported as a *safe import* (page 583) and that the *-fpackage-trust* (page 587) flag (see *flag* (page 585)) is used with packages from untrusted sources being marked as untrusted.

There is a more detailed discussion of the issues involved in compilation safety and some potential solutions on the GHC Wiki.

Additionally, the use of *annotations* (page 623) is forbidden, as that would allow bypassing Safe Haskell restrictions. See #10826 for details.

6.19 Miscellaneous

6.19.1 Rewrite rules

{-# RULES "⟨name⟩" forall ⟨binder⟩ ... . ⟨expr⟩ = ⟨expr⟩ ... #-}

Where top-level

Define a rewrite rule to be used to optimize a source program.

The programmer can specify rewrite rules as part of the source program (in a pragma). Here is an example:

```
{-# RULES
      "map/map"    forall f g xs.  map f (map g xs) = map (f . g) xs
  #-}
```

Use the debug flag *-ddump-simpl-stats* (page 258) to see what rules fired. If you need more information, then *-ddump-rule-firings* (page 258) shows you each individual rule firing and *-ddump-rule-rewrites* (page 258) also shows what the code looks like before and after the rewrite.

-fenable-rewrite-rules

Allow the compiler to apply rewrite rules to the source program.

6.19.1.1 Syntax

From a syntactic point of view:

- There may be zero or more rules in a *RULES* (page 589) pragma, separated by semicolons (which may be generated by the layout rule).
- The layout rule applies in a pragma. Currently no new indentation level is set, so if you put several rules in single RULES pragma and wish to use layout to separate them, you must lay out the starting in the same column as the enclosing definitions.

  ```
  {-# RULES
        "map/map"    forall f g xs.  map f (map g xs) = map (f . g) xs
        "map/append" forall f xs ys. map f (xs ++ ys) = map f xs ++ map f ys
    #-}
  ```

  Furthermore, the closing `#-}` should start in a column to the right of the opening `{-#`.
- Each rule has a name, enclosed in double quotes. The name itself has no significance at all. It is only used when reporting how many times the rule fired.
- A rule may optionally have a phase-control number (see *Phase control* (page 613)), immediately after the name of the rule. Thus:

  ```
  {-# RULES
        "map/map" [2]  forall f g xs. map f (map g xs) = map (f . g) xs
    #-}
  ```

  The `[2]` means that the rule is active in Phase 2 and subsequent phases. The inverse notation `[~2]` is also accepted, meaning that the rule is active up to, but not including, Phase 2.

Rules support the special phase-control notation `[~]`, which means the rule is never active. This feature supports plugins (see *Compiler Plugins* (page 625)), by making it possible to define a RULE that is never run by GHC, but is nevertheless parsed, typechecked etc, so that it is available to the plugin.

- Each (term) variable mentioned in a rule must either be in scope (e.g. `map`), or bound by the `forall` (e.g. `f`, `g`, `xs`). The variables bound by the `forall` are called the *pattern* variables. They are separated by spaces, just like in a type `forall`.
- A pattern variable may optionally have a type signature. If the type of the pattern variable is polymorphic, it *must* have a type signature. For example, here is the `foldr/build` rule:

```
"fold/build"  forall k z (g::forall b. (a->b->b) -> b -> b) .
              foldr k z (build g) = g k z
```

  Since `g` has a polymorphic type, it must have a type signature.

- If `ExplicitForAll` (page 506) is enabled, type/kind variables can also be explicitly bound. For example:

```
{-# RULES "id" forall a. forall (x :: a). id @a x = x #-}
```

  When a type-level explicit `forall` is present, each type/kind variable mentioned must now also be either in scope or bound by the `forall`. In particular, unlike some other places in Haskell, this means free kind variables will not be implicitly bound. For example:

```
"this_is_bad" forall (c :: k). forall (x :: Proxy c) ...
"this_is_ok"  forall k (c :: k). forall (x :: Proxy c) ...
```

  When bound type/kind variables are needed, both foralls must always be included, though if no pattern variables are needed, the second can be left empty. For example:

```
{-# RULES "map/id" forall a. forall. map (id @a) = id @[a] #-}
```

- The left hand side of a rule must consist of a top-level variable applied to arbitrary expressions. For example, this is *not* OK:

```
"wrong1"   forall e1 e2.  case True of { True -> e1; False -> e2 } = e1
"wrong2"   forall f.      f True = True
"wrong3"   forall x.      Just x = Nothing
```

  In `"wrong1"`, the LHS is not an application; in `"wrong2"`, the LHS has a pattern variable in the head. In `"wrong3"`, the LHS consists of a *constructor*, rather than a *variable*, applied to an argument.

- A rule does not need to be in the same module as (any of) the variables it mentions, though of course they need to be in scope.
- All rules are implicitly exported from the module, and are therefore in force in any module that imports the module that defined the rule, directly or indirectly. (That is, if A imports B, which imports C, then C's rules are in force when compiling A.) The situation is very similar to that for instance declarations.
- Inside a `RULES` (page 589) "`forall`" is treated as a keyword, regardless of any other flag settings. Furthermore, inside a `RULES` (page 589), the language extension `ScopedTypeVariables` (page 512) is automatically enabled; see *Lexically scoped type variables* (page 512).

- Like other pragmas, *RULES* (page 589) pragmas are always checked for scope errors, and are typechecked. Typechecking means that the LHS and RHS of a rule are typechecked, and must have the same type. However, rules are only *enabled* if the *-fenable-rewrite-rules* (page 589) flag is on (see *Semantics* (page 591)).

6.19.1.2 Semantics

From a semantic point of view:

- Rules are enabled (that is, used during optimisation) by the *-fenable-rewrite-rules* (page 589) flag. This flag is implied by *-O* (page 113), and may be switched off (as usual) by *-fno-enable-rewrite-rules* (page 589). (NB: enabling *-fenable-rewrite-rules* (page 589) without *-O* (page 113) may not do what you expect, though, because without *-O* (page 113) GHC ignores all optimisation information in interface files; see *-fignore-interface-pragmas* (page 118)). Note that *-fenable-rewrite-rules* (page 589) is an *optimisation* flag, and has no effect on parsing or typechecking.
- Rules are regarded as left-to-right rewrite rules. When GHC finds an expression that is a substitution instance of the LHS of a rule, it replaces the expression by the (appropriately-substituted) RHS. By "a substitution instance" we mean that the LHS can be made equal to the expression by substituting for the pattern variables.
- GHC makes absolutely no attempt to verify that the LHS and RHS of a rule have the same meaning. That is undecidable in general, and infeasible in most interesting cases. The responsibility is entirely the programmer's!
- GHC makes no attempt to make sure that the rules are confluent or terminating. For example:

  ```
  "loop"        forall x y.  f x y = f y x
  ```

  This rule will cause the compiler to go into an infinite loop.
- If more than one rule matches a call, GHC will choose one arbitrarily to apply.
- GHC currently uses a very simple, syntactic, matching algorithm for matching a rule LHS with an expression. It seeks a substitution which makes the LHS and expression syntactically equal modulo alpha conversion. The pattern (rule), but not the expression, is eta-expanded if necessary. (Eta-expanding the expression can lead to laziness bugs.) But not beta conversion (that's called higher-order matching).

  Matching is carried out on GHC's intermediate language, which includes type abstractions and applications. So a rule only matches if the types match too. See *Specialisation* (page 596) below.
- GHC keeps trying to apply the rules as it optimises the program. For example, consider:

  ```
  let s = map f
      t = map g
  in
  s (t xs)
  ```

  The expression `s (t xs)` does not match the rule `"map/map"`, but GHC will substitute for `s` and `t`, giving an expression which does match. If `s` or `t` was (a) used more than once, and (b) large or a redex, then it would not be substituted, and the rule would not fire.

- GHC will never match a forall'd variable in a template with an expression which contains locally bound variables. For example, it is permitted to write a rule which contains a case expression:

  ```
  {-# RULES
    "test/case-tup" forall (x :: (Int, Int)) (y :: Int) (z :: Int).
      test (case x of (l, r) -> y) z = case x of (l, r) -> test y z
    #-}
  ```

  But the rule will not match when `y` contains either of `l` or `r` because they are locally bound. Therefore the following application will fail to trigger the rule:

  ```
  prog :: (Int, Int) -> (Int, Int)
  prog x = test (case x of (p, q) -> p) 0
  ```

  because `y` would have to match against `p` (which is locally bound) but it will fire for:

  ```
  prog :: (Int, Int) -> (Int, Int)
  prog x = test (case x of (p, q) -> 0) 0
  ```

  because `y` can match against `0`.

- GHC implements **higher order matching** as described by GHC proposal #555. When a pattern variable is applied to distinct locally bound variables it forms what we call a **higher order pattern**. When matching, higher order patterns are treated like pattern variables, but they are allowed to match expressions that contain the locally bound variables that are part of the higher order patterns.

  For example, we can use this to fix the broken rule from the example from the previous bullet point:

  ```
  {-# RULES
    "test/case-tup" forall (x :: (Int, Int)) (f :: Int -> Int -> Int) (z :: Int).
      test (case x of (l, r) -> f l r) z = case x of (m, n) -> test (f m n) z
    #-}
  ```

  This modified rule does fire for:

  ```
  prog :: (Int, Int) -> (Int, Int)
  prog x = test (case x of (p, q) -> p) 0
  ```

  Under higher order matching, `f p q` matches `p` by assigning `f = \p q -> p`. The resulting code after the rewrite is:

  ```
  prog x = case x of (m, n) -> test ((\p q -> p) m n) 0
  ```

- A rule that has a forall binder with a polymorphic type, is likely to fail to fire. E. g.,

  ```
  {-# RULES forall (x :: forall a. Num a => a -> a).  f x = blah #-}
  ```

  Here `x` has a polymorphic type. This applies to a forall'd binder with a type class constraint, such as:

  ```
  {-# RULES forall @m (x :: KnownNat m => Proxy m).  g x = blah #-}
  ```

  See #21093 for discussion.

6.19.1.3 How rules interact with INLINE/NOINLINE pragmas

Ordinary inlining happens at the same time as rule rewriting, which may lead to unexpected results. Consider this (artificial) example

```
f x = x
g y = f y
h z = g True

{-# RULES "f" f True = False #-}
```

Since f's right-hand side is small, it is inlined into g, to give

```
g y = y
```

Now g is inlined into h, but f's RULE has no chance to fire. If instead GHC had first inlined g into h then there would have been a better chance that f's *RULES* (page 589) might fire.

The way to get predictable behaviour is to use a *NOINLINE* (page 612) pragma, or an INLINE[⟨phase⟩] pragma, on f, to ensure that it is not inlined until its *RULES* (page 589) have had a chance to fire. The warning flag *-Winline-rule-shadowing* (page 106) (see *Warnings and sanity-checking* (page 84)) warns about this situation.

6.19.1.4 How rules interact with CONLIKE pragmas

GHC is very cautious about duplicating work. For example, consider

```
f k z xs = let xs = build g
           in ...(foldr k z xs)...sum xs...
{-# RULES "foldr/build" forall k z g. foldr k z (build g) = g k z #-}
```

Since xs is used twice, GHC does not fire the foldr/build rule. Rightly so, because it might take a lot of work to compute xs, which would be duplicated if the rule fired.

Sometimes, however, this approach is over-cautious, and we *do* want the rule to fire, even though doing so would duplicate redex. There is no way that GHC can work out when this is a good idea, so we provide the CONLIKE pragma to declare it, thus:

```
{-# INLINE CONLIKE [1] f #-}
f x = blah
```

CONLIKE is a modifier to an INLINE or NOINLINE pragma. It specifies that an application of f to one argument (in general, the number of arguments to the left of the = sign) should be considered cheap enough to duplicate, if such a duplication would make rule fire. (The name "CONLIKE" is short for "constructor-like", because constructors certainly have such a property.) The *CONLIKE* (page 612) pragma is a modifier to *INLINE* (page 609)/*NOINLINE* (page 612) because it really only makes sense to match f on the LHS of a rule if you are sure that f is not going to be inlined before the rule has a chance to fire.

6.19.1.5 How rules interact with class methods

Giving a RULE for a class method is a bad idea:

```
class C a where
  op :: a -> a -> a

instance C Bool where
  op x y = ...rhs for op at Bool...

{-# RULES "f" op True y = False #-}
```

In this example, `op` is not an ordinary top-level function; it is a class method. GHC rapidly rewrites any occurrences of `op`-used-at-type-Bool to a specialised function, say `opBool`, where

```
opBool :: Bool -> Bool -> Bool
opBool x y = ..rhs for op at Bool...
```

So the RULE never has a chance to fire, for just the same reasons as in *How rules interact with INLINE/NOINLINE pragmas* (page 593).

The solution is to define the instance-specific function yourself, with a pragma to prevent it being inlined too early, and give a RULE for it:

```
instance C Bool where
  op = opBool

opBool :: Bool -> Bool -> Bool
{-# NOINLINE [1] opBool #-}
opBool x y = ..rhs for op at Bool...

{-# RULES "f" opBool True y = False #-}
```

If you want a RULE that truly applies to the overloaded class method, the only way to do it is like this:

```
class C a where
  op_c :: a -> a -> a

op :: C a => a -> a -> a
{-# NOINLINE [1] op #-}
op = op_c

{-# RULES "reassociate" op (op x y) z = op x (op y z) #-}
```

Now the inlining of `op` is delayed until the rule has a chance to fire. The down-side is that instance declarations must define `op_c`, but all other uses should go via `op`.

6.19.1.6 List fusion

The RULES mechanism is used to implement fusion (deforestation) of common list functions. If a "good consumer" consumes an intermediate list constructed by a "good producer", the intermediate list should be eliminated entirely.

The following are good producers:

- List comprehensions
- Enumerations of `Int`, `Integer` and `Char` (e.g. `['a'..'z']`).
- Explicit lists (e.g. `[True, False]`)
- The cons constructor (e.g `3:4:[]`)
- `++`
- `map`
- `take`, `filter`
- `iterate`, `repeat`
- `zip`, `zipWith`

The following are good consumers:

- List comprehensions
- `array` (on its second argument)
- `++` (on its first argument)
- `foldr`
- `map`
- `take`, `filter`
- `concat`
- `unzip`, `unzip2`, `unzip3`, `unzip4`
- `zip`, `zipWith` (but on one argument only; if both are good producers, `zip` will fuse with one but not the other)
- `partition`
- `head`
- `and`, `or`, `any`, `all`
- `sequence_`
- `msum`

So, for example, the following should generate no intermediate lists:

```
array (1,10) [(i,i*i) | i <- map (+ 1) [0..9]]
```

This list could readily be extended; if there are Prelude functions that you use a lot which are not included, please tell us.

If you want to write your own good consumers or producers, look at the Prelude definitions of the above functions to see how to do so.

6.19.1.7 Specialisation

Rewrite rules can be used to get the same effect as a feature present in earlier versions of GHC. For example, suppose that:

```
genericLookup :: Ord a => Table a b   -> a   -> b
intLookup     ::          Table Int b -> Int -> b
```

where `intLookup` is an implementation of `genericLookup` that works very fast for keys of type `Int`. You might wish to tell GHC to use `intLookup` instead of `genericLookup` whenever the latter was called with type `Table Int b -> Int -> b`. It used to be possible to write a *SPECIALIZE* (page 614) pragma with a right-hand-side:

```
{-# SPECIALIZE genericLookup :: Table Int b -> Int -> b = intLookup #-}
```

This feature is no longer in GHC, but rewrite rules let you do the same thing:

```
{-# RULES "genericLookup/Int" genericLookup = intLookup #-}
```

This slightly odd-looking rule instructs GHC to replace `genericLookup` by `intLookup` *whenever the types match*. What is more, this rule does not need to be in the same file as `genericLookup`, unlike the `SPECIALIZE` pragmas which currently do (so that they have an original definition available to specialise).

It is *Your Responsibility* to make sure that `intLookup` really behaves as a specialised version of `genericLookup`!!!

An example in which using `RULES` for specialisation will Win Big:

```
toDouble :: Real a => a -> Double
toDouble = fromRational . toRational

{-# RULES "toDouble/Int" toDouble = i2d #-}
i2d (I# i) = D# (int2Double# i) -- uses Glasgow prim-op directly
```

The `i2d` function is virtually one machine instruction; the default conversion—via an intermediate `Rational`-is obscenely expensive by comparison.

6.19.1.8 Controlling what's going on in rewrite rules

- Use *-ddump-rules* (page 258) to see the rules that are defined *in this module*. This includes rules generated by the specialisation pass, but excludes rules imported from other modules.
- Use *-ddump-simpl-stats* (page 258) to see what rules are being fired. If you add *-dppr-debug* (page 256) you get a more detailed listing.
- Use *-ddump-rule-firings* (page 258) or *-ddump-rule-rewrites* (page 258) to see in great detail what rules are being fired. If you add *-dppr-debug* (page 256) you get a still more detailed listing.
- The definition of (say) `build` in `GHC/Base.hs` looks like this:

  ```
  build   :: forall a. (forall b. (a -> b -> b) -> b -> b) -> [a]
  {-# INLINE build #-}
  build g = g (:) []
  ```

Notice the *INLINE* (page 609)! That prevents `(:)` from being inlined when compiling `PrelBase`, so that an importing module will "see" the `(:)`, and can match it on the LHS of a rule. `INLINE` prevents any inlining happening in the RHS of the `INLINE` thing. I regret the delicacy of this.

- In `libraries/base/GHC/Base.hs` look at the rules for `map` to see how to write rules that will do fusion and yet give an efficient program even if fusion doesn't happen. More rules in `GHC/List.hs`.

6.19.2 Special built-in functions

GHC has a few built-in functions with special behaviour. In particular:

- GHC.Exts.inline allows control over inlining on a per-call-site basis.
- GHC.Exts.lazy restrains the strictness analyser.
- GHC.Exts.oneShot gives a hint to the compiler about how often a function is being called.

6.19.3 Generic programming

There are a few ways to do datatype-generic programming using the GHC.Generics framework. One is making use of the `Generically` and `Generically1` wrappers from `GHC.Generics`, instances can be derived via them using *DerivingVia* (page 457):

```
{-# LANGUAGE DeriveGeneric      #-}
{-# LANGUAGE DerivingStrategies #-}
{-# LANGUAGE DerivingVia        #-}

import GHC.Generics

data V4 a = V4 a a a a
  deriving
  stock (Generic, Generic1)

  deriving (Semigroup, Monoid)
  via Generically (V4 a)

  deriving (Functor, Applicative)
  via Generically1 V4
```

The older approach uses *DeriveGeneric* (page 598), *DefaultSignatures* (page 472), and *DeriveAnyClass* (page 453). It derives instances by providing a distinguished generic implementation as part of the type class declaration. This section gives a very brief overview of how to do it.

Generic programming support in GHC allows defining classes with methods that do not need a user specification when instantiating: the method body is automatically derived by GHC. This is similar to what happens for standard classes such as `Read` and `Show`, for instance, but now for user-defined classes.

Note: GHC used to have an implementation of generic classes as defined in the paper "Derivable type classes", Ralf Hinze and Simon Peyton Jones, Haskell Workshop, Montreal Sept 2000, pp. 94-105. These have been removed and replaced by the more general support for generic programming.

6.19.3.1 Deriving representations

The first thing we need is generic representations. The `GHC.Generics` module defines a couple of primitive types that are used to represent Haskell datatypes:

```
-- | Unit: used for constructors without arguments
data U1 p = U1

-- | Constants, additional parameters and recursion of kind Type
newtype K1 i c p = K1 { unK1 :: c }

-- | Meta-information (constructor names, etc.)
newtype M1 i c f p = M1 { unM1 :: f p }

-- | Sums: encode choice between constructors
infixr 5 :+:
data (:+:) f g p = L1 (f p) | R1 (g p)

-- | Products: encode multiple arguments to constructors
infixr 6 :*:
data (:*:) f g p = f p :*: g p
```

The `Generic` and `Generic1` classes mediate between user-defined datatypes and their internal representation as a sum-of-products:

```
class Generic a where
  -- Encode the representation of a user datatype
  type Rep a :: Type -> Type
  -- Convert from the datatype to its representation
  from  :: a -> (Rep a) x
  -- Convert from the representation to the datatype
  to    :: (Rep a) x -> a

class Generic1 (f :: k -> Type) where
  type Rep1 f :: k -> Type

  from1  :: f a -> Rep1 f a
  to1    :: Rep1 f a -> f a
```

`Generic1` is used for functions that can only be defined over type containers, such as `map`. Note that `Generic1` ranges over types of kind `Type -> Type` by default, but if the *PolyKinds* (page 364) extension is enabled, then it can range of types of kind `k -> Type`, for any kind `k`.

DeriveGeneric

Since 7.2.1

Status Included in *GHC2024* (page 270), *GHC2021* (page 271)

Allow automatic deriving of instances for the `Generic` typeclass.

Instances of these classes can be derived by GHC with the *DeriveGeneric* (page 598) extension, and are necessary to be able to define generic instances automatically.

For example, a user-defined datatype of trees

```
data UserTree a = Node a (UserTree a) (UserTree a) | Leaf
```

in a `Main` module in a package named `foo` will get the following representation:

```
instance Generic (UserTree a) where
  -- Representation type
  type Rep (UserTree a) =
    M1 D ('MetaData "UserTree" "Main" "package-name" 'False) (
          M1 C ('MetaCons "Node" 'PrefixI 'False) (
                M1 S ('MetaSel 'Nothing
                               'NoSourceUnpackedness
                               'NoSourceStrictness
                               'DecidedLazy)
                     (K1 R a)
            :*: M1 S ('MetaSel 'Nothing
                               'NoSourceUnpackedness
                               'NoSourceStrictness
                               'DecidedLazy)
                     (K1 R (UserTree a))
            :*: M1 S ('MetaSel 'Nothing
                               'NoSourceUnpackedness
                               'NoSourceStrictness
                               'DecidedLazy)
                     (K1 R (UserTree a)))
      :+: M1 C ('MetaCons "Leaf" 'PrefixI 'False) U1)

  -- Conversion functions
  from (Node x l r) = M1 (L1 (M1 (M1 (K1 x) :*: M1 (K1 l) :*: M1 (K1 r))))
  from Leaf         = M1 (R1 (M1 U1))
  to (M1 (L1 (M1 (M1 (K1 x) :*: M1 (K1 l) :*: M1 (K1 r))))) = Node x l r
  to (M1 (R1 (M1 U1)))                                     = Leaf
```

This representation is generated automatically if a `deriving Generic` clause is attached to the datatype. *Standalone deriving* (page **??**) can also be used.

6.19.3.2 Writing generic functions

A generic function is defined by creating a class and giving instances for each of the representation types of `GHC.Generics`. As an example we show generic serialization:

```
data Bin = O | I

class GSerialize f where
  gput :: f a -> [Bin]

instance GSerialize U1 where
  gput U1 = []

instance (GSerialize a, GSerialize b) => GSerialize (a :*: b) where
  gput (x :*: y) = gput x ++ gput y

instance (GSerialize a, GSerialize b) => GSerialize (a :+: b) where
  gput (L1 x) = O : gput x
  gput (R1 x) = I : gput x

instance (GSerialize a) => GSerialize (M1 i c a) where
  gput (M1 x) = gput x

instance (Serialize a) => GSerialize (K1 i a) where
  gput (K1 x) = put x
```

A caveat: this encoding strategy may not be reliable across different versions of GHC. When deriving a `Generic` instance is free to choose any nesting of `:+:` and `:*:` it chooses, so if GHC chooses `(a :+: b) :+: c`, then the encoding for `a` would be `[O, O]`, `b` would be `[O, I]`, and `c` would be `[I]`. However, if GHC chooses `a :+: (b :+: c)`, then the encoding for `a` would be `[O]`, `b` would be `[I, O]`, and `c` would be `[I, I]`. (In practice, the current implementation tries to produce a more-or-less balanced nesting of `:+:` and `:*:` so that the traversal of the structure of the datatype from the root to a particular component can be performed in logarithmic rather than linear time.)

Typically this `GSerialize` class will not be exported, as it only makes sense to have instances for the representation types.

6.19.3.3 Unlifted representation types

The data family `URec` is provided to enable generic programming over datatypes with certain unlifted arguments. There are six instances corresponding to common unlifted types:

```
data family URec a p

data instance URec (Ptr ()) p = UAddr   { uAddr#   :: Addr#   }
data instance URec Char     p = UChar   { uChar#   :: Char#   }
data instance URec Double   p = UDouble { uDouble# :: Double# }
data instance URec Int      p = UInt    { uInt#    :: Int#    }
data instance URec Float    p = UFloat  { uFloat#  :: Float#  }
data instance URec Word     p = UWord   { uWord#   :: Word#   }
```

Six type synonyms are provided for convenience:

```
type UAddr   = URec (Ptr ())
type UChar   = URec Char
type UDouble = URec Double
type UFloat  = URec Float
type UInt    = URec Int
type UWord   = URec Word
```

As an example, this data declaration:

```
data IntHash = IntHash Int#
  deriving Generic
```

results in the following `Generic` instance:

```
instance 'Generic' IntHash where
  type 'Rep' IntHash =
    'D1' ('MetaData "IntHash" "Main" "package-name" 'False)
      ('C1' ('MetaCons "IntHash" 'PrefixI 'False)
        ('S1' ('MetaSel 'Nothing
                        'NoSourceUnpackedness
                        'NoSourceStrictness
                        'DecidedLazy)
              'UInt'))
```

A user could provide, for example, a `GSerialize UInt` instance so that a `Serialize IntHash` instance could be easily defined in terms of `GSerialize`.

6.19.3.4 Generic defaults

The only thing left to do now is to define a "front-end" class, which is exposed to the user:

```
class Serialize a where
  put :: a -> [Bin]

  default put :: (Generic a, GSerialize (Rep a)) => a -> [Bin]
  put = gput . from
```

Here we use a *default signature* (page **??**) to specify that the user does not have to provide an implementation for `put`, as long as there is a `Generic` instance for the type to instantiate. For the `UserTree` type, for instance, the user can just write:

```
instance (Serialize a) => Serialize (UserTree a)
```

The default method for `put` is then used, corresponding to the generic implementation of serialization. If you are using *DeriveAnyClass* (page 453), the same instance is generated by simply attaching a `deriving Serialize` clause to the `UserTree` datatype declaration. For more examples of generic functions please refer to the generic-deriving package on Hackage.

6.19.3.5 More information

For more details please refer to the Haskell Wiki page or the original paper [Generics2010].

6.19.4 Assertions

If you want to make use of assertions in your standard Haskell code, you could define a function like the following:

```
assert :: Bool -> a -> a
assert False x = error "assertion failed!"
assert _     x = x
```

which works, but gives you back a less than useful error message – an assertion failed, but which and where?

One way out is to define an extended `assert` function which also takes a descriptive string to include in the error message and perhaps combine this with the use of a pre-processor which inserts the source location where `assert` was used.

GHC offers a helping hand here, doing all of this for you. For every use of `assert` in the user's source:

```
kelvinToC :: Double -> Double
kelvinToC k = assert (k >= 0.0) (k-273.15)
```

GHC will rewrite this to also include the source location where the assertion was made,

```
assert pred val ==> assertError "Main.hs|15" pred val
```

The rewrite is only performed by the compiler when it spots applications of Control.Exception.assert, so you can still define and use your own versions of `assert`, should you so wish. If not, import `Control.Exception` to make use `assert` in your code.

GHC ignores assertions when optimisation is turned on with the *-O* (page 113) flag. That is, expressions of the form `assert pred e` will be rewritten to `e`. You can also disable assertions using the *-fignore-asserts* (page 118) option. The option *-fno-ignore-asserts* (page 118) allows enabling assertions even when optimisation is turned on.

Assertion failures can be caught, see the documentation for the Control.Exception library for the details.

6.19.5 HasCallStack

`GHC.Stack.HasCallStack` is a lightweight method of obtaining a partial call-stack at any point in the program.

A function can request its call-site with the `HasCallStack` constraint and access it as a Haskell value by using `callStack`.

One can then use functions from `GHC.Stack` to inspect or pretty print (as is done in `f` below) the call stack.

```
f :: HasCallStack => IO ()
f = putStrLn (prettyCallStack callStack)

g :: HasCallStack => IO ()
g = f
```

Evaluating `f` directly shows a call stack with a single entry, while evaluating `g`, which also requests its call-site, shows two entries, one for each computation "annotated" with `HasCallStack`.

```
ghci> f
CallStack (from HasCallStack):
  f, called at <interactive>:19:1 in interactive:Ghci1
ghci> g
CallStack (from HasCallStack):
  f, called at <interactive>:17:5 in main:Main
  g, called at <interactive>:20:1 in interactive:Ghci2
```

The `error` function from the Prelude supports printing the call stack that led to the error in addition to the usual error message:

```
ghci> error "bad"
*** Exception: bad
CallStack (from HasCallStack):
  error, called at <interactive>:25:1 in interactive:Ghci5
```

The call stack here consists of a single entry, pinpointing the source of the call to `error`. However, by annotating several computations with `HasCallStack`, figuring out the exact circumstances and sequences of calls that lead to a call to `error` becomes a lot easier, as demonstrated with the simple example below.

```
f :: HasCallStack => IO ()
f = error "bad bad bad"

g :: HasCallStack => IO ()
g = f
```

(continues on next page)

(continued from previous page)

```
h :: HasCallStack => IO ()
h = g
```

```
ghci> h
*** Exception: bad bad bad
CallStack (from HasCallStack):
  error, called at call-stack.hs:4:5 in main:Main
  f, called at call-stack.hs:7:5 in main:Main
  g, called at call-stack.hs:10:5 in main:Main
  h, called at <interactive>:28:1 in interactive:Ghci1
```

The `CallStack` will only extend as far as the types allow it, for example

```
myHead :: HasCallStack => [a] -> a
myHead []     = error "empty"
myHead (x:xs) = x

bad :: Int
bad = myHead []
```

```
ghci> bad
*** Exception: empty
CallStack (from HasCallStack):
  error, called at Bad.hs:8:15 in main:Bad
  myHead, called at Bad.hs:12:7 in main:Bad
```

includes the call-site of `error` in `myHead`, and of `myHead` in `bad`, but not the call-site of `bad` at the GHCi prompt.

GHC solves `HasCallStack` constraints in two steps:

1. If there is a `CallStack` in scope – i.e. the enclosing definition has a `HasCallStack` constraint – GHC will push the new call-site onto the existing `CallStack`.
2. Otherwise GHC will solve the `HasCallStack` constraint for the singleton `CallStack` containing just the current call-site.

Importantly, GHC will **never** infer a `HasCallStack` constraint, you must request it explicitly.

`CallStack` is kept abstract, but GHC provides a function

```
getCallStack :: CallStack -> [(String, SrcLoc)]
```

to access the individual call-sites in the stack. The `String` is the name of the function that was called, and the `SrcLoc` provides the package, module, and file name, as well as the line and column numbers.

`GHC.Stack` additionally exports a function `withFrozenCallStack` that allows users to freeze the current `CallStack`, preventing any future push operations from having an effect. This can be used by library authors to prevent `CallStack`s from exposing unnecessary implementation details. Consider the `myHead` example above, the `error` line in the printed stack is not particularly enlightening, so we might choose to suppress it by freezing the `CallStack` that we pass to `error`.

```
myHead :: HasCallStack => [a] -> a
myHead []     = withFrozenCallStack (error "empty")
myHead (x:xs) = x
```

```
ghci> myHead []
*** Exception: empty
CallStack (from HasCallStack):
  myHead, called at Bad.hs:12:7 in main:Bad
```

NOTE: The intrepid user may notice that `HasCallStack` is just an alias for an implicit parameter `?callStack :: CallStack`. This is an implementation detail and **should not** be considered part of the `CallStack` API, we may decide to change the implementation in the future.

6.19.5.1 Compared with other sources of stack traces

`HasCallStack` does not interact with the RTS and does not require compilation with `-prof`. On the other hand, as the `CallStack` is built up explicitly via the `HasCallStack` constraints, it will generally not contain as much information as the simulated call-stacks maintained by the RTS.

6.19.6 Whitespace

As in the Haskell Language Report, Haskell comments are valid whitespace. In addition, lines (which must end with a line feed character) that begin as follows are valid whitespace in source code, except immediately after a `where`, `let`, `do` or `of` keyword:

- `#!`. This accommodates 'shebang' interpreter directives in scripts on Unix-like operating systems.
- <space>#!, where <space> is an initial space character before the 'shebang'.
- `#pragma`. This accommodates the use of a directive that passes additional information to a compiler.
- #line <line> "<file>", where <line> is a positive integer and <file> can comprise zero or more characters. This accommodates a compiler directive that resets the numbering of lines of source code, and the identification of the source code file name, in compiler messages.
- #<line> "<file>", where <line> is a positive integer and <file> can comprise zero or more characters.

6.20 Pragmas

GHC supports several pragmas, or instructions to the compiler placed in the source code. Pragmas don't normally affect the meaning of the program, but they might affect the efficiency of the generated code.

Pragmas all take the form `{-# word ... #-}` where ⟨word⟩ indicates the type of pragma, and is followed optionally by information specific to that type of pragma. Case is ignored in ⟨word⟩. The various values for ⟨word⟩ that GHC understands are described in the following sections; any pragma encountered with an unrecognised ⟨word⟩ is ignored.

Certain pragmas are *file-header pragmas*:

- A file-header pragma must precede the `module` keyword in the file.

- There can be as many file-header pragmas as you please, and they can be preceded or followed by comments.
- File-header pragmas are read once only, before pre-processing the file (e.g. with cpp).
- The file-header pragmas are: `{-# LANGUAGE #-}`, `{-# OPTIONS_GHC #-}`, and `{-# INCLUDE #-}`.

6.20.1 LANGUAGE pragma

`{-# LANGUAGE ⟨ext⟩, ⟨ext⟩, ... #-}`

Where file header

Enable or disable a set of language extensions.

The `LANGUAGE` pragma allows language extensions to be enabled in a portable way. It is the intention that all Haskell compilers support the `LANGUAGE` pragma with the same syntax, although not all extensions are supported by all compilers, of course. The `LANGUAGE` pragma should be used instead of `OPTIONS_GHC`, if possible.

For example, to enable the FFI and preprocessing with CPP:

```
{-# LANGUAGE ForeignFunctionInterface, CPP #-}
```

`LANGUAGE` is a file-header pragma (see *Pragmas* (page 604)).

Every language extension can also be turned into a command-line flag by prefixing it with "`-X`"; for example `-XForeignFunctionInterface`. (Similarly, all "`-X`" flags can be written as `LANGUAGE` pragmas.)

A list of all supported language extensions can be obtained by invoking `ghc --supported-extensions` (see *`--supported-extensions`* (page 69)).

Any extension from the `Extension` type defined in Language.Haskell.Extension may be used. GHC will report an error if any of the requested extensions are not supported.

6.20.2 OPTIONS_GHC pragma

`{-# OPTIONS_GHC ⟨flags⟩ #-}`

Where file header

The `OPTIONS_GHC` pragma is used to specify additional options that are given to the compiler when compiling this source file. See *Command line options in source files* (page 66) for details.

Previous versions of GHC accepted `OPTIONS` rather than `OPTIONS_GHC`, but that is now deprecated.

`OPTIONS_GHC` is a file-header pragma (see *Pragmas* (page 604)).

6.20.3 INCLUDE pragma

The INCLUDE used to be necessary for specifying header files to be included when using the FFI and compiling via C. It is no longer required for GHC, but is accepted (and ignored) for compatibility with other compilers.

6.20.4 WARNING and DEPRECATED pragmas

{-# WARNING #-}

Where declaration or module name

The WARNING pragma allows you to attach an arbitrary warning to a particular function, class, type, export field or module.

{-# DEPRECATED #-}

Where declaration or module name

A DEPRECATED pragma lets you specify that a particular function, class, type, export or module is deprecated.

There are three ways of using these pragmas.

- You can work on an entire module thus:

```
module Wibble {-# DEPRECATED "Use Wobble instead" #-} where
  ...
```

  Or:

```
module Wibble {-# WARNING "This is an unstable interface." #-} where
  ...
```

  When you compile any module that import Wibble, GHC will print the specified message.

- You can attach a warning to a function, class, type, or data constructor, with the following top-level declarations:

```
{-# DEPRECATED f, C, T "Don't use these" #-}
{-# WARNING unsafePerformIO "This is unsafe; I hope you know what you're doing" #-
↪}
```

  When you compile any module that imports and uses any of the specified entities, GHC will print the specified message.

  You can only attach to entities declared at top level in the module being compiled, and you can only use unqualified names in the list of entities. A capitalised name, such as T refers to *either* the type constructor T *or* the data constructor T, or both if both are in scope. If both are in scope, there is currently no way to specify one without the other (c.f. fixities *Infix type constructors, classes, and type variables* (page 322)).

- You can add a warning to an instance (including derived instances):

```
instance {-# DEPRECATED "Don't use" #-} Show T1 where { .. }
instance {-# WARNING "Don't use either" #-} Show G1 where { .. }

deriving instance {-# DEPRECATED "to be removed" #-} Eq T2
deriving instance {-# WARNING "to be removed as well" #-} Eq G2
```

Doing so will cause warnings to be emitted whenever such instances are used to solve a constraint. For example:

```
foo = show (MkT1 :: T1) -- warning: uses "instance Show T1"

bar :: forall a. Eq a => a -> Bool
bar x = x == x
baz :: T2 -> Bool
baz = bar -- warning: uses "instance Eq T2"
quux :: Eq T2 => T2 -> Bool
quux = bar -- no warning: does not use "instance Eq T2"
```

As with other deprecation mechanisms, note that warnings will not be emitted for the usages of those instances in the module in which they are defined.

- Finally, you can attach a warning to an export field, be it a regular export:

```
module Wibble (
    {-# DEPRECATED "Do not use this type" #-} T,
    {-# WARNING "This is a hacky function" #-} f
  ) where
  ...
```

Or a re-export of import from another module:

```
module Wibble (
    {-# DEPRECATED "Import this function from A instead" #-} g
  ) where
import A
```

Or a re-export of an entire module:

```
module Wibble (
    {-# DEPRECATED "This declaration has been moved to B instead" #-}
      module B
  ) where
import B
```

When you compile any module that imports and uses any of the specified entities, GHC will print the specified message.

An entity will only be warned about if all of its exports are deprecated:

```
module Wibble (
    {-# WARNING "This would not be warned about" #-} g,
    module A
  )
import A (g)
```

If the :ghc-flag: *-Wincomplete-export-warnings* is on, such occurrences are warned about.

Moreover, all warning declarations of a specific name have to be warned with the same pragma and message:

```
module Wibble (
    {-# WARNING "This would throw an error" #-} T(T1),
    {-# WARNING "Because the warning messages differ for T" #-} T,
```

(continues on next page)

(continued from previous page)

```
)
...
```

Also note that the argument to DEPRECATED and WARNING can also be a list of strings, in which case the strings will be presented on separate lines in the resulting warning message,

```
{-# DEPRECATED foo, bar ["Don't use these", "Use gar instead"] #-}
```

Warnings and deprecations are not reported for (a) uses within the defining module, (b) defining a method in a class instance, (c) unqualified uses of an entity imported through different modules when not all of them are warned about, and (d) uses in an export list (except for export warnings). The latter reduces spurious complaints within a library in which one module gathers together and re-exports the exports of several others.

A WARNING pragma (but not a DEPRECATED pragma) may optionally specify a *warning category* as a string literal following the in keyword. This affects the flag used to suppress the warning. The examples above do not specify a category, so the default category deprecations applies, and they can be suppressed with the flag *-Wno-deprecations* (page 90) (and its synonym *-Wno-warnings-deprecations* (page 90)).

If a category is specified, the warning can instead be suppressed with the flag -Wno-x-⟨category⟩, for example warnings from the following pragma can be suppressed with -Wno-x-partial:

```
{-# WARNING in "x-partial" head "This function is partial..." #-}
```

Alternatively, warnings from all WARNING and DEPRECATED pragmas regardless of category can be suppressed with *-Wno-extended-warnings* (page 89).

When a deprecated name appears in both value and type namespaces (i.e. punning occurs) WARNING and DEPRECATED pragmas will affect both:

```
{-# LANGUAGE PatternSynonyms #-}

data D = MkD
pattern D = MkD
{-# DEPRECATED D "This will deprecate both the type D and the pattern synonym D" #-}
```

It is possible to specify the namespace of the name to be warned about or deprecated using type and data specifiers, but this feature requires enabling *ExplicitNamespaces* (page 319):

```
{-# LANGUAGE PatternSynonyms #-}

data D = MkD
pattern D = MkD
{-# DEPRECATED data D "This will deprecate only the pattern synonym D" #-}
{-# DEPRECATED type D "This will deprecate only the type D" #-}
```

6.20.5 MINIMAL pragma

{-# MINIMAL ⟨name⟩ | ⟨name⟩ , ... #-}

Where in class body

Define the methods needed for a minimal complete instance of a class.

The `MINIMAL` pragma is used to specify the minimal complete definition of a class, i.e. specify which methods must be implemented by all instances. If an instance does not satisfy the minimal complete definition, then a warning is generated. This can be useful when a class has methods with circular defaults. For example

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
    {-# MINIMAL (==) | (/=) #-}
```

Without the `MINIMAL` pragma no warning would be generated for an instance that implements neither method.

The syntax for minimal complete definition is:

```
mindef ::= name
        |  '(' mindef ')'
        |  mindef '|' mindef
        |  mindef ',' mindef
```

A vertical bar denotes disjunction, i.e. one of the two sides is required. A comma denotes conjunction, i.e. both sides are required. Conjunction binds stronger than disjunction.

If no `MINIMAL` pragma is given in the class declaration, it is just as if a pragma `{-# MINIMAL op1, op2, ..., opn #-}` was given, where the `opi` are the methods that lack a default method in the class declaration (c.f. *-Wmissing-methods* (page 97), *Warnings and sanity-checking* (page 84)).

This warning can be turned off with the flag *-Wno-missing-methods* (page 97).

6.20.6 INLINE and NOINLINE pragmas

These pragmas control the inlining of function definitions.

6.20.6.1 INLINE pragma

{-# INLINE ⟨name⟩ #-}

Where any function definition

Force GHC to inline a value.

GHC (with *-O* (page 113), as always) tries to inline (or "unfold") functions/values that are "small enough," thus avoiding the call overhead and possibly exposing other more-wonderful optimisations. GHC has a set of heuristics, tuned over a long period of time using many benchmarks, that decide when it is beneficial to inline a function at its call site. The heuristics are designed to inline functions when it appears to be beneficial to do so, but without incurring excessive code bloat. If a function looks too big, it won't be inlined, and functions larger than a certain size will not even have their definition exported in the interface file. Some of the thresholds that govern these heuristic decisions can be changed using flags, see *-f*: platform-independent flags* (page 113).

Normally GHC will do a reasonable job of deciding by itself when it is a good idea to inline a function. However, sometimes you might want to override the default behaviour. For example, if you have a key function that is important to inline because it leads to further optimisations, but GHC judges it to be too big to inline.

The sledgehammer you can bring to bear is the `INLINE` pragma, used thusly:

```
key_function :: Int -> String -> (Bool, Double)
{-# INLINE key_function #-}
```

The major effect of an `INLINE` pragma is to declare a function's "cost" to be very low. The normal unfolding machinery will then be very keen to inline it. However, an `INLINE` pragma for a function "`f`" has a number of other effects:

- While GHC is keen to inline the function, it does not do so blindly. For example, if you write

  ```
  map key_function xs
  ```

  there really isn't any point in inlining `key_function` to get

  ```
  map (\x -> body) xs
  ```

  In general, GHC only inlines the function if there is some reason (no matter how slight) to suppose that it is useful to do so.

- Moreover, GHC will only inline the function if it is *fully applied*, where "fully applied" means applied to as many arguments as appear (syntactically) on the LHS of the function definition. For example:

  ```
  comp1 :: (b -> c) -> (a -> b) -> a -> c
  {-# INLINE comp1 #-}
  comp1 f g = \x -> f (g x)

  comp2 :: (b -> c) -> (a -> b) -> a -> c
  {-# INLINE comp2 #-}
  comp2 f g x = f (g x)
  ```

  The two functions `comp1` and `comp2` have the same semantics, but `comp1` will be inlined when applied to *two* arguments, while `comp2` requires *three*. This might make a big difference if you say

  ```
  map (not `comp1` not) xs
  ```

  which will optimise better than the corresponding use of `comp2`.

- It is useful for GHC to optimise the definition of an INLINE function `f` just like any other non-`INLINE` function, in case the non-inlined version of `f` is ultimately called. But we don't want to inline the *optimised* version of `f`; a major reason for `INLINE` pragmas is to expose functions in `f`'s RHS that have rewrite rules, and it's no good if those functions have been optimised away.

  So *GHC guarantees to inline precisely the code that you wrote*, no more and no less. It does this by capturing a copy of the definition of the function to use for inlining (we call this the "inline-RHS"), which it leaves untouched, while optimising the ordinarily

RHS as usual. For externally-visible functions the inline-RHS (not the optimised RHS) is recorded in the interface file.

- An `INLINE` function is not worker/wrappered by strictness analysis. It's going to be inlined wholesale instead.

GHC ensures that inlining cannot go on forever: every mutually-recursive group is cut by one or more *loop breakers* that is never inlined (see Secrets of the GHC inliner, JFP 12(4) July 2002). GHC tries not to select a function with an `INLINE` pragma as a loop breaker, but when there is no choice even an INLINE function can be selected, in which case the `INLINE` pragma is ignored. For example, for a self-recursive function, the loop breaker can only be the function itself, so an `INLINE` pragma is always ignored.

`INLINE` pragmas are a particularly good idea for the `then`/`return` (or `bind`/`unit`) functions in a monad. For example, in GHC's own `UniqueSupply` monad code, we have:

```
{-# INLINE thenUs #-}
{-# INLINE returnUs #-}
```

See also the `NOINLINE` (*NOINLINE pragma* (page 612)) and `INLINABLE` (*INLINABLE pragma* (page 612)) pragmas.

`INLINE` pragma effects on various locations

Syntactically, an `INLINE` pragma for a function can be put anywhere its type signature could be put. This means a `INLINE` pragma can really be put on any definition site for a binding. This includes top-level, `let` and `where` bindings as well as default class methods and instance declarations.

The pragma itself will only have an effect when the RHS of the binding it's applied to is used. For regular bindings this is straight forward but for class methods and instance definitions this can have surprising ramifications.

If we consider a class definition with two instances like this:

```
class C a where
    op1 :: a -> a

    op2 :: [a] -> [a]
    op2 xs = reverse (xs ++ xs)
    {-# INLINE op2 #-}

instance C T1 where
    op1 x = ...blah...

instance C T2 where
    {-# INLINE op1 #-}
    op1 x = ...blah...
    op2 xs = ...blah...
```

Then `op2` for the T1 instance will get an implicit `INLINE` pragma. This is because the RHS of the default method is used for `op2` which retains it's `INLINE` pragma.

In the T2 instance `op1` gets an `INLINE` pragma and behaves accordingly. However `op2` for T2 is **not** implemented by the default method. This means the pragma in the class definition doesn't apply to this instance. With no pragma being explicitly applied GHC will then decide on a proper inlining behaviour for `T2`s `op2` method on it's own.

6.20.6.2 `INLINABLE` pragma

`{-# INLINABLE ⟨name⟩ #-}`

Where any function definition

Suggest that the compiler always consider inlining `name`.

An `{-# INLINABLE f #-}` pragma on a function `f` has the following behaviour:

- While `INLINE` says "please inline me", the `INLINABLE` says "feel free to inline me; use your discretion". In other words the choice is left to GHC, which uses the same rules as for pragma-free functions. Unlike `INLINE`, that decision is made at the *call site*, and will therefore be affected by the inlining threshold, optimisation level etc.
- Like `INLINE`, the `INLINABLE` pragma retains a copy of the original RHS for inlining purposes, and persists it in the interface file, regardless of the size of the RHS.
- One way to use `INLINABLE` is in conjunction with the special function `inline` (*Special built-in functions* (page 597)). The call `inline f` tries very hard to inline `f`. To make sure that `f` can be inlined, it is a good idea to mark the definition of `f` as `INLINABLE`, so that GHC guarantees to expose an unfolding regardless of how big it is. Moreover, by annotating `f` as `INLINABLE`, you ensure that `f`'s original RHS is inlined, rather than whatever random optimised version of `f` GHC's optimiser has produced.
- The `INLINABLE` pragma also works with `SPECIALISE`: if you mark function `f` as `INLINABLE`, then you can subsequently `SPECIALISE` in another module (see *SPECIALIZE pragma* (page 614)).
- Unlike `INLINE`, it is OK to use an `INLINABLE` pragma on a recursive function. The principal reason do to so to allow later use of `SPECIALISE`

The alternative spelling `INLINEABLE` is also accepted by GHC.

6.20.6.3 `NOINLINE` pragma

`{-# NOINLINE ⟨name⟩ #-}`

Where any function definition

Instructs the compiler not to inline a value.

The *`NOINLINE`* (page 612) pragma does exactly what you'd expect: it stops the named function from being inlined by the compiler. You shouldn't ever need to do this, unless you're very cautious about code size.

`NOTINLINE` is a synonym for `NOINLINE` (`NOINLINE` is specified by Haskell 98 as the standard way to disable inlining, so it should be used if you want your code to be portable).

6.20.6.4 `CONLIKE` modifier

`{-# CONLIKE #-}`

Where modifies *`INLINE`* (page 609) or *`NOINLINE`* (page 612) pragma

Instructs GHC to consider a value to be especially cheap to inline.

An *`INLINE`* (page 609) or *`NOINLINE`* (page 612) pragma may have a *`CONLIKE`* (page 612) modifier, which affects matching in *`RULE`* (page 589)s (only). See *How rules interact with CONLIKE pragmas* (page 593).

6.20.6.5 Phase control

Sometimes you want to control exactly when in GHC's pipeline the *INLINE* (page 609) pragma is switched on. Inlining happens only during runs of the *simplifier*. Each run of the simplifier has a different *phase number*; the phase number decreases towards zero. If you use *-dverbose-core2core* (page 258) you will see the sequence of phase numbers for successive runs of the simplifier. In an *INLINE* (page 609) pragma you can optionally specify a phase number, thus:

- "`INLINE[k] f`" means: do not inline `f` until phase `k`, but from phase `k` onwards be very keen to inline it.
- "`INLINE[~k] f`" means: be very keen to inline `f` until phase `k`, but from phase `k` onwards do not inline it.
- "`NOINLINE[k] f`" means: do not inline `f` until phase `k`, but from phase `k` onwards be willing to inline it (as if there was no pragma).
- "`NOINLINE[~k] f`" means: be willing to inline `f` until phase `k`, but from phase `k` onwards do not inline it.

The same information is summarised here:

```
                         -- Before phase 2     Phase 2 and later
{-# INLINE   [2]  f #-}  --      No                  Yes
{-# INLINE   [~2] f #-}  --      Yes                 No
{-# NOINLINE [2]  f #-}  --      No                  Maybe
{-# NOINLINE [~2] f #-}  --      Maybe               No

{-# INLINE   f #-}       --      Yes                 Yes
{-# NOINLINE f #-}       --      No                  No
```

By "Maybe" we mean that the usual heuristic inlining rules apply (if the function body is small, or it is applied to interesting-looking arguments etc). Another way to understand the semantics is this:

- For both *INLINE* (page 609) and *NOINLINE* (page 612), the phase number says when inlining is allowed at all.
- The *INLINE* (page 609) pragma has the additional effect of making the function body look small, so that when inlining is allowed it is very likely to happen.

The same phase-numbering control is available for *RULE* (page 589)s (*Rewrite rules* (page 589)).

6.20.7 `OPAQUE` pragma

`{-# OPAQUE ⟨name⟩ #-}`

Where top-level

Instructs the compiler to ensure that every call of `name` remains a call of `name`, and not some name-mangled variant.

The *OPAQUE* (page 613) pragma is an even stronger variant of the *NOINLINE* (page 612) pragma. Like the *NOINLINE* (page 612), named functions annotated with a *OPAQUE* (page 613) pragma are not inlined, nor will they be be specialized. Unlike the *NOINLINE* (page 612), named functions annotated with a *OPAQUE* (page 613) pragma are left untouched by the Worker/Wrapper transformation. Unlike *NOINLINE* (page 612), *OPAQUE* (page 613) has no phase control.

In effect, every call of a named function annotated with an *OPAQUE* (page 613) pragma remains a call of that named function, not some name-mangled variant. You shouldn't ever need to use the *OPAQUE* (page 613) pragma, unless you have a reason to care about name-mangling.

6.20.8 LINE pragma

{-# LINE ⟨lineno⟩ "⟨file⟩" #-}

Where anywhere

Generated by preprocessors to convey source line numbers of the original source.

This pragma is similar to C's `#line` pragma, and is mainly for use in automatically generated Haskell code. It lets you specify the line number and filename of the original code; for example

```
{-# LINE 42 "Foo.vhs" #-}
```

if you'd generated the current file from something called `Foo.vhs` and this line corresponds to line 42 in the original. GHC will adjust its error messages to refer to the line/file named in the `LINE` pragma.

`LINE` pragmas generated from Template Haskell set the file and line position for the duration of the splice and are limited to the splice. Note that because Template Haskell splices abstract syntax, the file positions are not automatically advanced.

6.20.9 COLUMN pragma

This is the analogue of the `LINE` pragma and is likewise intended for use in automatically generated Haskell code. It lets you specify the column number of the original code; for example

```
foo = do
  {-# COLUMN 42 #-}pure ()
  pure ()
```

This adjusts all column numbers immediately after the pragma to start at 42. The presence of this pragma only affects the quality of the diagnostics and does not change the syntax of the code itself.

6.20.10 RULES pragma

The *RULES* (page 589) pragma lets you specify rewrite rules. It is described in *Rewrite rules* (page 589).

6.20.11 SPECIALIZE pragma

{-# SPECIALIZE ⟨name⟩ :: ⟨type⟩ #-}

Ask that GHC specialize a polymorphic value to a particular type.

(UK spelling also accepted.) For key overloaded functions, you can create extra versions (NB: at the cost of larger code) specialised to particular types. Thus, if you have an overloaded function:

```
hammeredLookup :: Ord key => [(key, value)] -> key -> value
```

If it is heavily used on lists with `Widget` keys, you could specialise it as follows:

```
{-# SPECIALIZE hammeredLookup :: [(Widget, value)] -> Widget -> value #-}
```

- A `SPECIALIZE` pragma for a function can be put anywhere its type signature could be put. Moreover, you can also `SPECIALIZE` an *imported* function provided it was given an `INLINABLE` pragma at its definition site (*INLINABLE pragma* (page 612)).
- A `SPECIALIZE` has the effect of generating (a) a specialised version of the function and (b) a rewrite rule (see *Rewrite rules* (page 589)) that rewrites a call to the un-specialised function into a call to the specialised one. Moreover, given a `SPECIALIZE` pragma for a function `f`, GHC will automatically create specialisations for any type-class-overloaded functions called by `f`, if they are in the same module as the `SPECIALIZE` pragma, or if they are `INLINABLE`; and so on, transitively.
- You can add phase control (*Phase control* (page 613)) to the RULE generated by a `SPECIALIZE` pragma, just as you can if you write a `RULE` directly. For example:

  ```
  {-# SPECIALIZE [0] hammeredLookup :: [(Widget, value)] -> Widget -> value #-}
  ```

  generates a specialisation rule that only fires in Phase 0 (the final phase). If you do not specify any phase control in the `SPECIALIZE` pragma, the phase control is inherited from the inline pragma (if any) of the function. For example:

  ```
  foo :: Num a => a -> a
  foo = ...blah...
  {-# NOINLINE [0] foo #-}
  {-# SPECIALIZE foo :: Int -> Int #-}
  ```

  The `NOINLINE` pragma tells GHC not to inline `foo` until Phase 0; and this property is inherited by the specialisation RULE, which will therefore only fire in Phase 0.

  The main reason for using phase control on specialisations is so that you can write optimisation RULES that fire early in the compilation pipeline, and only *then* specialise the calls to the function. If specialisation is done too early, the optimisation rules might fail to fire.
- The type in a `SPECIALIZE` pragma can be any type that is less polymorphic than the type of the original function. In concrete terms, if the original function is `f` then the pragma

  ```
  {-# SPECIALIZE f :: <type> #-}
  ```

  is valid if and only if the definition

  ```
  f_spec :: <type>
  f_spec = f
  ```

  is valid. Here are some examples (where we only give the type signature for the original function, not its code):

  ```
  f :: Eq a => a -> b -> b
  {-# SPECIALISE f :: Int -> b -> b #-}

  g :: (Eq a, Ix b) => a -> b -> b
  {-# SPECIALISE g :: (Eq a) => a -> Int -> Int #-}

  h :: Eq a => a -> a -> a
  {-# SPECIALISE h :: (Eq a) => [a] -> [a] -> [a] #-}
  ```

The last of these examples will generate a RULE with a somewhat-complex left-hand side (try it yourself), so it might not fire very well. If you use this kind of specialisation, let us know how well it works.

6.20.11.1 SPECIALIZE INLINE

{-# SPECIALIZE INLINE ⟨name⟩ :: ⟨type⟩ #-}

Where top-level

A `SPECIALIZE` pragma can optionally be followed with a `INLINE` or `NOINLINE` pragma, optionally followed by a phase, as described in *INLINE and NOINLINE pragmas* (page 609). The `INLINE` pragma affects the specialised version of the function (only), and applies even if the function is recursive. The motivating example is this:

```
-- A GADT for arrays with type-indexed representation
data Arr e where
  ArrInt :: !Int -> ByteArray# -> Arr Int
  ArrPair :: !Int -> Arr e1 -> Arr e2 -> Arr (e1, e2)

(!:) :: Arr e -> Int -> e
{-# SPECIALISE INLINE (!:) :: Arr Int -> Int -> Int #-}
{-# SPECIALISE INLINE (!:) :: Arr (a, b) -> Int -> (a, b) #-}
(ArrInt _ ba)     !: (I# i) = I# (indexIntArray# ba i)
(ArrPair _ a1 a2) !: i      = (a1 !: i, a2 !: i)
```

Here, `(!:)` is a recursive function that indexes arrays of type `Arr e`. Consider a call to `(!:)` at type `(Int,Int)`. The second specialisation will fire, and the specialised function will be inlined. It has two calls to `(!:)`, both at type `Int`. Both these calls fire the first specialisation, whose body is also inlined. The result is a type-based unrolling of the indexing function.

You can add explicit phase control (*Phase control* (page 613)) to `SPECIALISE INLINE` pragma, just like on an *INLINE* (page 609) pragma; if you do so, the same phase is used for the rewrite rule and the INLINE control of the specialised function.

> **Warning:** You can make GHC diverge by using `SPECIALISE INLINE` on an ordinarily-recursive function.

6.20.11.2 SPECIALIZE for imported functions

Generally, you can only give a *SPECIALIZE* (page 614) pragma for a function defined in the same module. However if a function `f` is given an *INLINABLE* (page 612) pragma at its definition site, then it can subsequently be specialised by importing modules (see *INLINABLE pragma* (page 612)). For example

```
module Map( lookup, blah blah ) where
  lookup :: Ord key => [(key,a)] -> key -> Maybe a
  lookup = ...
  {-# INLINABLE lookup #-}

module Client where
  import Map( lookup )
```

(continues on next page)

(continued from previous page)

```
  data T = T1 | T2 deriving( Eq, Ord )
  {-# SPECIALISE lookup :: [(T,a)] -> T -> Maybe a
```

Here, `lookup` is declared *INLINABLE* (page 612), but it cannot be specialised for type `T` at its definition site, because that type does not exist yet. Instead a client module can define `T` and then specialise `lookup` at that type.

Moreover, every module that imports `Client` (or imports a module that imports `Client`, transitively) will "see", and make use of, the specialised version of `lookup`. You don't need to put a *SPECIALIZE* (page 614) pragma in every module.

Moreover you often don't even need the *SPECIALIZE* (page 614) pragma in the first place. When compiling a module `M`, GHC's optimiser (when given the *-O* (page 113) flag) automatically considers each top-level overloaded function declared in `M`, and specialises it for the different types at which it is called in `M`. The optimiser *also* considers each *imported* *INLINABLE* (page 612) overloaded function, and specialises it for the different types at which it is called in `M`. So in our example, it would be enough for `lookup` to be called at type `T`:

```
module Client where
  import Map( lookup )

  data T = T1 | T2 deriving( Eq, Ord )

  findT1 :: [(T,a)] -> Maybe a
  findT1 m = lookup m T1   -- A call of lookup at type T
```

However, sometimes there are no such calls, in which case the pragma can be useful.

6.20.12 SPECIALIZE instance pragma

{-# SPECIALIZE instance ⟨instance head⟩ #-}

Where instance body

Same idea, except for instance declarations. For example:

```
instance (Eq a) => Eq (Foo a) where {
   {-# SPECIALIZE instance Eq (Foo [(Int, Bar)]) #-}
   ... usual stuff ...
 }
```

The pragma must occur inside the `where` part of the instance declaration.

6.20.13 UNPACK pragma

{-# UNPACK #-}

Where data constructor field

Instructs the compiler to unpack the contents of a constructor field into the constructor itself.

The `UNPACK` indicates to the compiler that it should unpack the contents of a constructor field into the constructor itself, removing a level of indirection. For example:

```
data T = T {-# UNPACK #-} !Float
           {-# UNPACK #-} !Float
```

will create a constructor T containing two unboxed floats. This may not always be an optimisation: if the T constructor is scrutinised and the floats passed to a non-strict function for example, they will have to be reboxed (this is done automatically by the compiler).

Unpacking constructor fields should only be used in conjunction with *-O* (page 113)[1], in order to expose unfoldings to the compiler so the reboxing can be removed as often as possible. For example:

```
f :: T -> Float
f (T f1 f2) = f1 + f2
```

The compiler will avoid reboxing `f1` and `f2` by inlining `+` on floats, but only when *-O* (page 113) is on.

Any single-constructor data is eligible for unpacking; for example

```
data T = T {-# UNPACK #-} !(Int,Int)
```

will store the two `Int`s directly in the T constructor, by flattening the pair. Multi-level unpacking is also supported:

```
data T = T {-# UNPACK #-} !S
data S = S {-# UNPACK #-} !Int {-# UNPACK #-} !Int
```

will store two unboxed `Int#`s directly in the `T` constructor. The unpacker can see through newtypes, too.

Since 9.6.1, data types with multiple constructors can also be unpacked, effectively transforming the field into an unboxed sum of the unpackings of each constructor (see *UnboxedSums* (page 557)).

See also the *-funbox-strict-fields* (page 129) flag, which essentially has the effect of adding `{-# UNPACK #-}` to every strict constructor field which is of a single-constructor data type. Sum types won't be unpacked automatically by this though, only with the explicit pragma.

6.20.14 `NOUNPACK` pragma

{-# NOUNPACK #-}

Where top-level

Instructs the compiler not to unpack a constructor field.

The `NOUNPACK` pragma indicates to the compiler that it should not unpack the contents of a constructor field. Example:

```
data T = T {-# NOUNPACK #-} !(Int,Int)
```

Even with the flags *-funbox-strict-fields* (page 129) and *-O* (page 113), the field of the constructor T is not unpacked.

[1] In fact, *UNPACK* (page 617) has no effect without *-O* (page 113), for technical reasons (see #5252).

6.20.15 SOURCE pragma

{-# SOURCE #-}

Where after `import` statement

Import a module by `hs-boot` file to break a module loop.

The `{-# SOURCE #-}` pragma is used only in `import` declarations, to break a module loop. It is described in detail in *How to compile mutually recursive modules* (page 207).

6.20.16 COMPLETE pragmas

{-# COMPLETE #-}

Where at top level

Specify the set of constructors or pattern synonyms which constitute a total match.

The `COMPLETE` pragma is used to inform the pattern match checker that a certain set of patterns is complete and that any function which matches on all the specified patterns is total.

The most common usage of `COMPLETE` pragmas is with *Pattern synonyms* (page 461). On its own, the checker is very naive and assumes that any match involving a pattern synonym will fail. As a result, any pattern match on a pattern synonym is regarded as incomplete unless the user adds a catch-all case.

For example, the data types `2 * A` and `A + A` are isomorphic but some computations are more naturally expressed in terms of one or the other. To get the best of both worlds, we can choose one as our implementation and then provide a set of pattern synonyms so that users can use the other representation if they desire. We can then specify a `COMPLETE` pragma in order to inform the pattern match checker that a function which matches on both `LeftChoice` and `RightChoice` is total.

```
data Choice a = Choice Bool a

pattern LeftChoice :: a -> Choice a
pattern LeftChoice a = Choice False a

pattern RightChoice :: a -> Choice a
pattern RightChoice a = Choice True a

{-# COMPLETE LeftChoice, RightChoice #-}

foo :: Choice Int -> Int
foo (LeftChoice n) = n * 2
foo (RightChoice n) = n - 2
```

`COMPLETE` pragmas are only used by the pattern match checker. If a function definition matches on all the constructors specified in the pragma then the compiler will produce no warning.

`COMPLETE` pragmas can contain any data constructors or pattern synonyms which are in scope, but must mention at least one data constructor or pattern synonym defined in the same module. `COMPLETE` pragmas may only appear at the top level of a module. Once defined, they are automatically imported and exported from modules. `COMPLETE` pragmas should be thought of as asserting a universal truth about a set of patterns and as a result, should not be used to silence context specific incomplete match warnings.

It is also possible to restrict the types to which a `COMPLETE` pragma applies by putting a double colon `::` after the list of constructors, followed by a result type constructor, which will be used to restrict the cases in which the pragma applies. GHC will compare the annotated result type constructor with the type constructor in the head of the scrutinee type in a pattern match to see if the `COMPLETE` pragma is meant to apply to it.

This is especially useful in cases that the constructors specified are polymorphic, e.g.:

```
data Proxy a = Proxy

class IsEmpty a where
  isEmpty :: a -> Bool

class IsCons a where
  type Elt a
  isCons :: a -> Maybe (Elt a, a)

pattern Empty :: IsEmpty a => a
pattern Empty <- (isEmpty -> True)

pattern Cons :: IsCons a => Elt a -> a -> a
pattern Cons x xs <- (isCons -> Just (x,xs))

instance IsEmpty (Proxy a) where
  isEmpty Proxy = True

instance IsEmpty [a] where
  isEmpty = null

instance IsCons [a] where
  type Elt [a] = a
  isCons [] = Nothing
  isCons (x:xs) = Just (x,xs)

{-# COMPLETE Empty :: Proxy #-}
{-# COMPLETE Empty, Cons :: [] #-}

foo :: Proxy a -> Int
foo Empty = 0

bar :: [a] -> Int
bar Empty = 0
bar (Cons _ _) = 1

baz :: [a] -> Int
baz Empty = 0
```

In this example, `foo` and `bar` will not be warned about, as their pattern matches are covered by the two `COMPLETE` pragmas above, but `baz` will be warned about as incomplete.

6.20.17 OVERLAPPING, OVERLAPPABLE, OVERLAPS, and INCOHERENT pragmas

{-# OVERLAPPING #-}

{-# OVERLAPPABLE #-}

{-# OVERLAPS #-}

{-# INCOHERENT #-}

Where on instance head

The pragmas OVERLAPPING, OVERLAPPABLE, OVERLAPS, INCOHERENT are used to specify the overlap behavior for individual instances, as described in Section *Overlapping instances* (page 486). The pragmas are written immediately after the instance keyword, like this:

```
instance {-# OVERLAPPING #-} C t where ...
```

CHAPTER

SEVEN

EXTENDING AND USING GHC AS A LIBRARY

GHC exposes its internal APIs to users through the built-in ghc package. It allows you to write programs that leverage GHC's entire compilation driver, in order to analyze or compile Haskell code programmatically. Furthermore, GHC gives users the ability to load compiler plugins during compilation - modules which are allowed to view and change GHC's internal intermediate representation, Core. Plugins are suitable for things like experimental optimizations or analysis, and offer a lower barrier of entry to compiler development for many common cases.

Furthermore, GHC offers a lightweight annotation mechanism that you can use to annotate your source code with metadata, which you can later inspect with either the compiler API or a compiler plugin.

7.1 Source annotations

Annotations are small pragmas that allow you to attach data to identifiers in source code, which are persisted when compiled. These pieces of data can then inspected and utilized when using GHC as a library or writing a compiler plugin.

7.1.1 Annotating values

Any expression that has both `Typeable` and `Data` instances may be attached to a top-level value binding using an `ANN` pragma. In particular, this means you can use `ANN` to annotate data constructors (e.g. `Just`) as well as normal values (e.g. `take`). By way of example, to annotate the function `foo` with the annotation `Just "Hello"` you would do this:

```
{-# ANN foo (Just "Hello") #-}
foo = ...
```

A number of restrictions apply to use of annotations:

- The binder being annotated must be at the top level (i.e. no nested binders)
- The binder being annotated must be declared in the current module
- The expression you are annotating with must have a type with `Typeable` and `Data` instances
- The *Template Haskell staging restrictions* (page 534) apply to the expression being annotated with, so for example you cannot run a function from the module being compiled.

> To be precise, the annotation `{-# ANN x e #-}` is well staged if and only if `$(e)` would be (disregarding the usual type restrictions of the splice syntax, and the usual restriction on splicing inside a splice - `$([|1|])` is fine as an annotation, albeit redundant).

If you feel strongly that any of these restrictions are too onerous, please give the GHC team a shout.

However, apart from these restrictions, many things are allowed, including expressions which are not fully evaluated! Annotation expressions will be evaluated by the compiler just like Template Haskell splices are. So, this annotation is fine:

```
{-# ANN f SillyAnnotation { foo = (id 10) + $([| 20 |]), bar = 'f } #-}
f = ...
```

7.1.2 Annotating types

You can annotate types with the `ANN` pragma by using the `type` keyword. For example:

```
{-# ANN type Foo (Just "A `Maybe String' annotation") #-}
data Foo = ...
```

7.1.3 Annotating modules

You can annotate modules with the `ANN` pragma by using the `module` keyword. For example:

```
{-# ANN module (Just "A `Maybe String' annotation") #-}
```

7.2 Using GHC as a Library

The `ghc` package exposes most of GHC's frontend to users, and thus allows you to write programs that leverage it. This library is actually the same library used by GHC's internal, frontend compilation driver, and thus allows you to write tools that programmatically compile source code and inspect it. Such functionality is useful in order to write things like IDE or refactoring tools. As a simple example, here's a program which compiles a module, much like ghc itself does by default when invoked:

```
import GHC
import GHC.Paths ( libdir )
import GHC.Driver.Session ( defaultFatalMessager, defaultFlushOut )

main =
    defaultErrorHandler defaultFatalMessager defaultFlushOut $ do
      runGhc (Just libdir) $ do
        dflags <- getSessionDynFlags
        setSessionDynFlags dflags
        target <- guessTarget "test_main.hs" Nothing
        setTargets [target]
        load LoadAllTargets
```

The argument to `runGhc` is a bit tricky. GHC needs this to find its libraries, so the argument must refer to the directory that is printed by `ghc --print-libdir` for the same version of

GHC that the program is being compiled with. Above we therefore use the `ghc-paths` package which provides this for us.

Compiling it results in:

```
$ cat test_main.hs
main = putStrLn "hi"
$ ghc -package ghc simple_ghc_api.hs
[1 of 1] Compiling Main             ( simple_ghc_api.hs, simple_ghc_api.o )
Linking simple_ghc_api ...
$ ./simple_ghc_api
$ ./test_main
hi
$
```

For more information on using the API, as well as more samples and references, please see this Haskell.org wiki page.

7.3 Compiler Plugins

GHC has the ability to load compiler plugins at compile time. The feature is similar to the one provided by GCC, and allows users to write plugins that can adjust the behaviour of the constraint solver, inspect and modify the compilation pipeline, as well as transform and inspect GHC's intermediate language, Core. Plugins are suitable for experimental analysis or optimization, and require no changes to GHC's source code to use.

Plugins cannot optimize/inspect C--, nor can they implement things like parser/front-end modifications like GCC, apart from limited changes to the constraint solver. If you feel strongly that any of these restrictions are too onerous, please give the GHC team a shout.

Plugins do not work with `-fexternal-interpreter`. If you need to run plugins with `-fexternal-interpreter` let GHC developers know in #14335.

7.3.1 Using compiler plugins

Plugins can be added on the command line with the *-fplugin=⟨module⟩* (page 625) option where ⟨module⟩ is a module in a registered package that exports the plugin. Plugins are loaded in order, with command-line and Cabal flags preceding those in OPTIONS pragmas which are processed in file order. Arguments can be passed to the plugins with the *-fplugin-opt=⟨module⟩:⟨args⟩* (page 625) option. The list of enabled plugins can be reset with the *-fclear-plugins* (page 626) option.

-fplugin=⟨module⟩
: Load the plugin in the given module. The module must be a member of a package registered in GHC's package database.

-fplugin-opt=⟨module⟩:⟨args⟩
: Give arguments to a plugin module; module must be specified with *-fplugin=⟨module⟩* (page 625). The order of plugin pragmas matter but the order of arg pragmas does not. The same set of arguments go to all plugins from the same module.

```
-- Two Echo plugins will both get args A and B.
{-# OPTIONS -fplugin Echo -fplugin-opt Echo:A #-}
{-# OPTIONS -fplugin Echo -fplugin-opt Echo:B #-}
```

(continues on next page)

(continued from previous page)

```
-- While order of the plugins matters, arg order does not.
{-# OPTIONS -fplugin-opt Echo2:B #-}

{-# OPTIONS -fplugin Echo1 #-}
{-# OPTIONS -fplugin-opt Echo1:A #-}

{-# OPTIONS -fplugin Echo2 #-}
```

If you want to use the same plugin with different arguments then rexport the same plugin from different lightweight modules.

```
-- Echo1 and Echo2 as lightweight modules re-exporting Echo.plugin.
module Echo1 (plugin) where import Echo (plugin)
module Echo2 (plugin) where import Echo (plugin)

-- Echo1 gets arg A while Echo2 gets arg B.
{-# OPTIONS -fplugin Echo1 -fplugin-opt Echo1:A #-}
{-# OPTIONS -fplugin Echo2 -fplugin-opt Echo2:B #-}
```

-fplugin-trustworthy

By default, when a module is compiled with plugins, it will be marked as unsafe. With this flag passed, all plugins are treated as trustworthy and the safety inference will no longer be affected.

-fclear-plugins

Clear the list of plugins previously specified with *-fplugin* (page 625). This is useful in GHCi where simply removing the *-fplugin* (page 625) options from the command line is not possible. Instead `:set -fclear-plugins` can be used.

As an example, in order to load the plugin exported by `Foo.Plugin` in the package `foo-ghc-plugin`, and give it the parameter "baz", we would invoke GHC like this:

```
$ ghc -fplugin Foo.Plugin -fplugin-opt Foo.Plugin:baz Test.hs
[1 of 1] Compiling Main             ( Test.hs, Test.o )
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Loading package foo-ghc-plugin-0.1 ... linking ... done.
...
Linking Test ...
$
```

Plugins can be also be loaded from libraries directly. It allows plugins to be loaded in cross-compilers (as a workaround for #14335).

-fplugin-library=⟨file-path⟩;⟨unit-id⟩;⟨module⟩;⟨args⟩

Arguments are specified in a list form, so a plugin specified to *-fplugin-library=⟨file-path⟩;⟨unit-id⟩;⟨module⟩;⟨args⟩* (page 626) will look like `'path/to/plugin;package-123;Plugin.Module;["Argument","List"]'`.

Alternatively, core plugins can be specified with Template Haskell.

```
addCorePlugin "Foo.Plugin"
```

This inserts the plugin as a core-to-core pass. Unlike *-fplugin=(module)*, the plugin module can't reside in the same package as the module calling Language.Haskell.TH.Syntax.addCorePlugin. This way, the implementation can expect the plugin to be built by the time it is needed.

Plugin modules live in a separate namespace from the user import namespace. By default, these two namespaces are the same; however, there are a few command line options which control specifically plugin packages:

-plugin-package ⟨pkg⟩
: This option causes the installed package ⟨pkg⟩ to be exposed for plugins, such as *-fplugin=⟨module⟩* (page 625). The package ⟨pkg⟩ can be specified in full with its version number (e.g. `network-1.0`) or the version number can be omitted if there is only one version of the package installed. If there are multiple versions of ⟨pkg⟩ installed and *-hide-all-plugin-packages* (page 627) was not specified, then all other versions will become hidden. *-plugin-package ⟨pkg⟩* (page 627) supports thinning and renaming described in *Thinning and renaming modules* (page 223).

  Unlike *-package ⟨pkg⟩* (page 221), this option does NOT cause package ⟨pkg⟩ to be linked into the resulting executable or shared object.

-plugin-package-id ⟨pkg-id⟩
: Exposes a package in the plugin namespace like *-plugin-package ⟨pkg⟩* (page 627), but the package is named by its installed package ID rather than by name. This is a more robust way to name packages, and can be used to select packages that would otherwise be shadowed. Cabal passes *-plugin-package-id ⟨pkg-id⟩* (page 627) flags to GHC. *-plugin-package-id ⟨pkg-id⟩* (page 627) supports thinning and renaming described in *Thinning and renaming modules* (page 223).

-hide-all-plugin-packages
: By default, all exposed packages in the normal, source import namespace are also available for plugins. This causes those packages to be hidden by default. If you use this flag, then any packages with plugins you require need to be explicitly exposed using *-plugin-package ⟨pkg⟩* (page 627) options.

At the moment, the only way to specify a dependency on a plugin in Cabal is to put it in `build-depends` (which uses the conventional *-package-id ⟨unit-id⟩* (page 221) flag); however, in the future there will be a separate field for specifying plugin dependencies specifically.

7.3.2 Writing compiler plugins

Plugins are modules that export at least a single identifier, `plugin`, of type `GHC.Plugins.Plugin`. All plugins should `import GHC.Plugins` as it defines the interface to the compilation pipeline.

A `Plugin` effectively holds a function which installs a compilation pass into the compiler pipeline. By default there is the empty plugin which does nothing, `GHC.Plugins.defaultPlugin`, which you should override with record syntax to specify your installation function. Since the exact fields of the `Plugin` type are open to change, this is the best way to ensure your plugins will continue to work in the future with minimal interface impact.

`Plugin` exports a field, `installCoreToDos` which is a function of type `[CommandLineOption] -> [CoreToDo] -> CoreM [CoreToDo]`. A `CommandLineOption` is effectively just `String`, and a `CoreToDo` is basically a function of type `Core -> Core`. A `CoreToDo` gives your pass a name and runs it over every compiled module when you invoke GHC.

As a quick example, here is a simple plugin that just does nothing and just returns the original compilation pipeline, unmodified, and says 'Hello':

```haskell
module DoNothing.Plugin (plugin) where
import GHC.Plugins

plugin :: Plugin
plugin = defaultPlugin {
  installCoreToDos = install
  }

install :: [CommandLineOption] -> [CoreToDo] -> CoreM [CoreToDo]
install _ todo = do
  putMsgS "Hello!"
  return todo
```

Provided you compiled this plugin and registered it in a package (with cabal for instance,) you can then use it by just specifying `-fplugin=DoNothing.Plugin` on the command line, and during the compilation you should see GHC say 'Hello'.

Running multiple plugins is also supported, by passing multiple `-fplugin=...` options. GHC will load the plugins in the order in which they are specified on the command line and, when appropriate, compose their effects in the same order. That is, if we had two Core plugins, `Plugin1` and `Plugin2`, each defining an `install` function like the one above, then GHC would first run `Plugin1.install` on the default `[CoreToDo]`, take the result and feed it to `Plugin2.install`. `-fplugin=Plugin1 -fplugin=Plugin2` will update the Core pipeline by applying `Plugin1.install opts1 >=> Plugin2.install opts2` (where `opts1` and `opts2` are the options passed to each plugin using `-fplugin-opt=...`). This is not specific to Core plugins but holds for all the types of plugins that can be composed or sequenced in some way: the first plugin to appear on the GHC command line will always act first.

7.3.3 Core plugins in more detail

`CoreToDo` is effectively a data type that describes all the kinds of optimization passes GHC does on Core. There are passes for simplification, CSE, etc. There is a specific case for plugins, `CoreDoPluginPass :: String -> PluginPass -> CoreToDo` which should be what you always use when inserting your own pass into the pipeline. The first parameter is the name of the plugin, and the second is the pass you wish to insert.

`CoreM` is a monad that all of the Core optimizations live and operate inside of.

A plugin's installation function (`install` in the above example) takes a list of `CoreToDo`s and returns a list of `CoreToDo`. Before GHC begins compiling modules, it enumerates all the needed plugins you tell it to load, and runs all of their installation functions, initially on a list of passes that GHC specifies itself. After doing this for every plugin, the final list of passes is given to the optimizer, and are run by simply going over the list in order.

You should be careful with your installation function, because the list of passes you give back isn't questioned or double checked by GHC at the time of this writing. An installation function like the following:

```haskell
install :: [CommandLineOption] -> [CoreToDo] -> CoreM [CoreToDo]
install _ _ = return []
```

is certainly valid, but also certainly not what anyone really wants.

7.3.3.1 Manipulating bindings

In the last section we saw that besides a name, a `CoreDoPluginPass` takes a pass of type `PluginPass`. A `PluginPass` is a synonym for `(ModGuts -> CoreM ModGuts)`. `ModGuts` is a type that represents the one module being compiled by GHC at any given time.

A `ModGuts` holds all of the module's top level bindings which we can examine. These bindings are of type `CoreBind` and effectively represent the binding of a name to body of code. Top-level module bindings are part of a `ModGuts` in the field `mg_binds`. Implementing a pass that manipulates the top level bindings merely needs to iterate over this field, and return a new `ModGuts` with an updated `mg_binds` field. Because this is such a common case, there is a function provided named `bindsOnlyPass` which lifts a function of type `([CoreBind] -> CoreM [CoreBind])` to type `(ModGuts -> CoreM ModGuts)`.

Continuing with our example from the last section, we can write a simple plugin that just prints out the name of all the non-recursive bindings in a module it compiles:

```
module SayNames.Plugin (plugin) where
import GHC.Plugins

plugin :: Plugin
plugin = defaultPlugin {
  installCoreToDos = install
  }

install :: [CommandLineOption] -> [CoreToDo] -> CoreM [CoreToDo]
install _ todo = do
  return (CoreDoPluginPass "Say name" pass : todo)

pass :: ModGuts -> CoreM ModGuts
pass guts = do dflags <- getDynFlags
               bindsOnlyPass (mapM (printBind dflags)) guts
  where printBind :: DynFlags -> CoreBind -> CoreM CoreBind
        printBind dflags bndr@(NonRec b _) = do
          putMsgS $ "Non-recursive binding named " ++ showSDoc dflags (ppr b)
          return bndr
        printBind _ bndr = return bndr
```

7.3.3.2 Late Plugins

If the `CoreProgram` of a module is modified in a normal core plugin, the modified bindings can end up in unfoldings the interface file for the module. This may be undesireable, as the plugin could make changes which affect inlining or optimization.

Late plugins can be used to avoid introducing such changes into the interface file. Late plugins are a bit different from typical core plugins:

1. They do not run in the `CoreM` monad. Instead, they are explicitly passed the `HscEnv` and they run in `IO`.
2. They are given `CgGuts` instead of `ModGuts`. `CgGuts` are a restricted form of `ModGuts` intended for code generation. The `CoreProgram` held in the `CgGuts` given to a late plugin will already be fully optimized.
3. They must maintain a `CostCentreState` and track any cost centres they introduce by adding them to the `cg_ccs` field of `CgGuts`. This is because the automatic collection of cost centres happens before the late plugin stage. If a late plugin does not introduce any cost centres, it may simply return the given cost centre state.

Here is a very simply example of a late plugin that changes the value of a binding in a module. If it finds a non-recursive top-level binding named `testBinding` with type `Int`, it will change its value to the `Int` expression `111111`.

```
plugin :: Plugin
plugin = defaultPlugin { latePlugin = lateP }

lateP :: LatePlugin
lateP _ _ (cg_guts, cc_state) = do
    binds' <- editCoreBinding (cg_binds cg_guts)
    return (cg_guts { cg_binds = binds' }, cc_state)

editCoreBinding :: CoreProgram -> IO CoreProgram
editCoreBinding pgm = pure . go
  where
    go :: [CoreBind] -> [CoreBind]
    go (b@(NonRec v e) : bs)
      | occNameString (getOccName v) == "testBinding" && exprType e `eqType` intTy =
          NonRec v (mkUncheckedIntExpr 111111) : bs
    go (b:bs) = b : go bs
    go [] = []
```

Since this is a late plugin, the changed binding value will not end up in the interface file.

7.3.3.3 Using Annotations

Previously we discussed annotation pragmas (*Source annotations* (page 623)), which we mentioned could be used to give compiler plugins extra guidance or information. Annotations for a module can be retrieved by a plugin, but you must go through the modules `ModGuts` in order to get it. Because annotations can be arbitrary instances of `Data` and `Typeable`, you need to give a type annotation specifying the proper type of data to retrieve from the interface file, and you need to make sure the annotation type used by your users is the same one your plugin uses. For this reason, we advise distributing annotations as part of the package which also provides compiler plugins if possible.

To get the annotations of a single binder, you can use `getAnnotations` and specify the proper type. Here’s an example that will print out the name of any top-level non-recursive binding with the `SomeAnn` annotation:

```
{-# LANGUAGE DeriveDataTypeable #-}
module SayAnnNames.Plugin (plugin, SomeAnn(..)) where
import GHC.Plugins
import Control.Monad (unless)
import Data.Data

data SomeAnn = SomeAnn deriving Data

plugin :: Plugin
plugin = defaultPlugin {
  installCoreToDos = install
  }

install :: [CommandLineOption] -> [CoreToDo] -> CoreM [CoreToDo]
install _ todo = do
  return (CoreDoPluginPass "Say name" pass : todo)
```

(continues on next page)

(continued from previous page)

```
pass :: ModGuts -> CoreM ModGuts
pass g = do
          dflags <- getDynFlags
          mapM_ (printAnn dflags g) (mg_binds g) >> return g
  where printAnn :: DynFlags -> ModGuts -> CoreBind -> CoreM CoreBind
        printAnn dflags guts bndr@(NonRec b _) = do
          anns <- annotationsOn guts b :: CoreM [SomeAnn]
          unless (null anns) $ putMsgS $ "Annotated binding found: " ++  showSDoc␣
↪dflags (ppr b)
          return bndr
        printAnn _ _ bndr = return bndr

annotationsOn :: Data a => ModGuts -> CoreBndr -> CoreM [a]
annotationsOn guts bndr = do
  (_, anns) <- getAnnotations deserializeWithData guts
  return $ lookupWithDefaultUFM_Directly anns [] (varUnique bndr)
```

Please see the GHC API documentation for more about how to use internal APIs, etc.

7.3.4 Typechecker plugins

In addition to Core plugins, GHC has experimental support for typechecker plugins, which allow the behaviour of the constraint solver to be modified. For example, they make it possible to interface the compiler to an SMT solver, in order to support a richer theory of type-level arithmetic expressions than the theory built into GHC (see *Computing With Type-Level Naturals* (page 386)).

The `Plugin` type has a field `tcPlugin` of type `[CommandLineOption] -> Maybe TcPlugin`, where the `TcPlugin` type is defined thus:

```
  data TcPlugin = forall s . TcPlugin
    { tcPluginInit    :: TcPluginM s
    , tcPluginSolve   :: s -> TcPluginSolver
    , tcPluginRewrite :: s -> UniqFM TyCon TcPluginRewriter
    , tcPluginStop    :: s -> TcPluginM ()
    }

  type TcPluginSolver = EvBindsVar -> [Ct] -> [Ct] -> TcPluginM TcPluginSolveResult

  type TcPluginRewriter = RewriteEnv -> [Ct] -> [Type] -> TcPluginM␣
↪TcPluginRewriteResult

data TcPluginSolveResult
  = TcPluginSolveResult
      { tcPluginInsolubleCts :: [Ct]
      , tcPluginSolvedCts    :: [(EvTerm, Ct)]
      , tcPluginNewCts       :: [Ct]
      }

  data TcPluginRewriteResult
    = TcPluginNoRewrite
    | TcPluginRewriteTo
        { tcPluginRewriteTo    :: Reduction
        , tcRewriterNewWanteds :: [Ct]
        }
```

(The details of this representation are subject to change as we gain more experience writing typechecker plugins. It should not be assumed to be stable between GHC releases.)

The basic idea is as follows:

- When type checking a module, GHC calls `tcPluginInit` once before constraint solving starts. This allows the plugin to look things up in the context, initialise mutable state or open a connection to an external process (e.g. an external SMT solver). The plugin can return a result of any type it likes, and the result will be passed to the other fields of the `TcPlugin` record.
- During constraint solving, GHC repeatedly calls `tcPluginSolve`. This function is provided with the current set of constraints, and should return a `TcPluginSolveResult` that indicates whether a contradiction was found or progress was made. If the plugin solver makes progress, GHC will re-start the constraint solving pipeline, looping until a fixed point is reached.
- When rewriting type family applications, GHC calls `tcPluginRewriter`. The plugin supplies a collection of type families which it is interested in rewriting. For each of those, the rewriter is provided with the the arguments to that type family, as well as the current collection of Given constraints. The plugin can then specify a rewriting for this type family application, if desired.
- Finally, GHC calls `tcPluginStop` after constraint solving is finished, allowing the plugin to dispose of any resources it has allocated (e.g. terminating the SMT solver process).

Plugin code runs in the `TcPluginM` monad, which provides a restricted interface to GHC API functionality that is relevant for typechecker plugins, including `IO` and reading the environment. If you need functionality that is not exposed in the `TcPluginM` module, you can use `unsafeTcPluginTcM :: TcM a -> TcPluginM a`, but are encouraged to contact the GHC team to suggest additions to the interface. Note that `TcPluginM` can perform arbitrary IO via `tcPluginIO :: IO a -> TcPluginM a`, although some care must be taken with side effects (particularly in `tcPluginSolve`). In general, it is up to the plugin author to make sure that any IO they do is safe.

7.3.4.1 Constraint solving with plugins

The key component of a typechecker plugin is a function of type `TcPluginSolver`, like this:

```
solve :: EvBindsVar -> [Ct] -> [Ct] -> TcPluginM TcPluginSolveResult
solve binds givens wanteds = ...
```

This function will be invoked in two different ways:

1. after simplification of Given constraints, where the plugin gets the opportunity to rewrite givens,
2. after GHC has attempted to solve Wanted constraints.

The two ways can be distinguished by checking the Wanted constraints: in the first case (and the first case only), the plugin will be passed an empty list of Wanted constraints.

The plugin can then respond with:

- solved constraints, which will be removed from the inert set,
- new constraints, which will be added to the work list,
- insoluble constraints, which will be reported as errors.

The plugin must respond with constraints of the same flavour, i.e. in (1) it should return only Givens, and for (2) it should return only Wanteds; all other constraints will be ignored.

If the plugin cannot make any progress, it should return `TcPluginSolveResult [] [] []`. Otherwise, if there were any new constraints, the main constraint solver will be re-invoked to simplify them, then the plugin will be invoked again. The plugin is responsible for making sure that this process eventually terminates.

Plugins are provided with all available constraints (including equalities and typeclass constraints), but it is easy for them to discard those that are not relevant to their domain, because they need return only those constraints for which they have made progress (either by solving or contradicting them).

Constraints that have been solved by the plugin must be provided with evidence in the form of an `EvTerm` of the type of the constraint. This evidence is ignored for Given constraints, which GHC "solves" simply by discarding them; typically this is used when they are uninformative (e.g. reflexive equations). For Wanted constraints, the evidence will form part of the Core term that is generated after typechecking, and can be checked by `-dcore-lint`. It is possible for the plugin to create equality axioms for use in evidence terms, but GHC does not check their consistency, and inconsistent axiom sets may lead to segfaults or other runtime misbehaviour.

Evidence is required also when creating new Given constraints, which are usually implied by old ones. It is not uncommon that the evidence of a new Given constraint contains a removed constraint: the new one has replaced the removed one.

7.3.4.2 Type family rewriting with plugins

Typechecker plugins can also directly rewrite type family applications, by supplying the `tcPluginRewrite` field of the `TcPlugin` record.

```
tcPluginRewrite :: s -> UniqFM TyCon TcPluginRewriter
```

That is, the plugin registers a map, from a type family's `TyCon` to its associated rewriting function:

```
type TcPluginRewriter = [Ct] -> [Type] -> TcPluginM TcPluginRewriteResult
```

This rewriting function is supplied with the Given constraints from the current context, and the type family arguments. Note that the type family application is guaranteed to be exactly saturated. This function should then return a possible rewriting of the type family application, by means of the following datatype:

```
data TcPluginRewriteResult
  = TcPluginNoRewrite
  | TcPluginRewriteTo
      { tcPluginRewriteTo    :: Reduction
      , tcRewriterNewWanteds :: [Ct]
      }
```

That is, the rewriter can specify a rewriting of the type family application – in which case it can also emit new Wanted constraints – or it can do nothing.

To specify a rewriting, the plugin must provide a `Reduction`, which is defined as follows:

```
data Reduction = Reduction Coercion !Type
```

That is, on top of specifying what type the type-family application rewrites to, the plugin must also supply a coercion which witnesses this rewriting:

```
co :: F orig_arg_1 ... orig_arg_n ~ rewritten_ty
```

Note in particular that the LHS type of the coercion should be the original type-family application, while its RHS type is the type that the plugin wants to rewrite the type-family application to.

7.3.5 Source plugins

In addition to core and type checker plugins, you can install plugins that can access different representations of the source code. The main purpose of these plugins is to make it easier to implement development tools.

There are several different access points that you can use for defining plugins that access the representations. All these fields receive the list of `CommandLineOption` strings that are passed to the compiler using the *-fplugin-opt=⟨module⟩:⟨args⟩* (page 625) flags.

```
plugin :: Plugin
plugin = defaultPlugin {
    parsedResultAction = parsed
  , typeCheckResultAction = typechecked
  , spliceRunAction = spliceRun
  , interfaceLoadAction = interfaceLoad
  , renamedResultAction = renamed
  }
```

7.3.5.1 Parsed representation

When you want to define a plugin that uses the syntax tree of the source code, you would like to override the `parsedResultAction` field. This access point enables you to get access to information about the lexical tokens and comments in the source code as well as the original syntax tree of the compiled module.

```
parsed :: [CommandLineOption] -> ModSummary
            -> ParsedResult -> Hsc ParsedResult
```

The `ModSummary` contains useful meta-information about the compiled module. The `ParsedResult` contains a `HsParsedModule`, which contains the lexical and syntactical information we mentioned before. The result that you return will change the result of the parsing. If you don't want to change the result, just return the `ParsedResult` that you received as the argument.

If the parser encounters any errors that prevent an AST from being constructed, the plugin will not be run, but other kinds of errors, as well as warnings, will be given to the plugin via the `PsMessages` value of the `ParsedResult`. This allows you to modify, remove, and add warnings or errors before they are displayed to the user, although in most cases, you will likely want to return the messages unmodified. The parsing pass will fail if the `Messages PsError` collection inside the return `ParsedResult` is not empty after all parsing plugins have been run.

7.3.5.2 Type checked representation

When you want to define a plugin that needs semantic information about the source code, use the typeCheckResultAction field. For example, if your plugin have to decide if two names are referencing the same definition or it has to check the type of a function it is using semantic information. In this case you need to access the renamed or type checked version of the syntax tree with typeCheckResultAction or renamedResultAction.

```
typechecked :: [CommandLineOption] -> ModSummary -> TcGblEnv -> TcM TcGblEnv
renamed :: [CommandLineOption] -> TcGblEnv -> HsGroup GhcRn -> TcM (TcGblEnv, HsGroup␣
↪GhcRn)
```

By overriding the renamedResultAction field we can modify each HsGroup after it has been renamed. A source file is separated into groups depending on the location of template haskell splices so the contents of these groups may not be intuitive. In order to save the entire renamed AST for inspection at the end of typechecking you can set renamedResultAction to keepRenamedSource which is provided by the Plugins module. This is important because some parts of the renamed syntax tree (for example, imports) are not found in the typechecked one.

7.3.5.3 Evaluated code

When the compiler type checks the source code, *Template Haskell* (page 527) Splices and *Template Haskell Quasi-quotation* (page 538) will be replaced by the syntax tree fragments generated from them. However for tools that operate on the source code the code generator is usually more interesting than the generated code. For this reason we included spliceRunAction. This field is invoked on each expression before they are evaluated. The input is type checked, so semantic information is available for these syntax tree fragments. If you return a different expression you can change the code that is generated.

```
spliceRun :: [CommandLineOption] -> LHsExpr GhcTc -> TcM (LHsExpr GhcTc)
```

However take care that the generated definitions are still in the input of typeCheckResultAction. If your don't take care to filter the typechecked input, the behavior of your tool might be inconsistent.

7.3.5.4 Interface files

Sometimes when you are writing a tool, knowing the source code is not enough, you also have to know details about the modules that you import. In this case we suggest using the interfaceLoadAction. This will be called each time when the code of an already compiled module is loaded. It will be invoked for modules from installed packages and even modules that are installed with GHC. It will NOT be invoked with your own modules.

```
interfaceLoad :: forall lcl . [CommandLineOption] -> ModIface
                               -> IfM lcl ModIface
```

In the ModIface datatype you can find lots of useful information, including the exported definitions and type class instances.

The ModIface datatype also contains facilities for extending it with extra data, stored in a Map of serialised fields, indexed by field names and using GHC's internal Binary class. The interface to work with these fields is:

```
readIfaceField :: Binary a => FieldName -> ModIface -> IO (Maybe a)
writeIfaceField :: Binary a => FieldName -> a -> ModIface -> IO ModIface
deleteIfaceField :: FieldName -> ModIface -> ModIface
```

The `FieldName` is open-ended, but typically it should contain the producing package name, along with the actual field name. Then, the version number can either be attached to the serialised data for that field, or in cases where multiple versions of a field could exist in the same interface file, included in the field name.

Depending on if the field version advances with the package version, or independently, the version can be attached to either the package name or the field name. Examples of each case:

```
package/field
ghc-n.n.n/core
package/field-n
```

To read an interface file from an external tool without linking to GHC, the format is described at Extensible Interface Files.

7.3.5.5 Source plugin example

In this example, we inspect all available details of the compiled source code. We don't change any of the representation, but write out the details to the standard output. The pretty printed representation of the parsed, renamed and type checked syntax tree will be in the output as well as the evaluated splices and quasi quotes. The name of the interfaces that are loaded will also be displayed.

```
module SourcePlugin where

import Control.Monad.IO.Class
import GHC.Driver.Session (getDynFlags)
import GHC.Driver.Plugins
import GHC.Plugins
import GHC.Tc.Types
import Language.Haskell.Syntax.Extension
import GHC.Hs.Decls
import GHC.Hs.Expr
import GHC.Hs.ImpExp
import GHC.Types.Avail
import GHC.Utils.Outputable
import GHC.Hs.Doc
import GHC

plugin :: Plugin
plugin = defaultPlugin
  { parsedResultAction = parsedPlugin
  , renamedResultAction = renamedAction
  , typeCheckResultAction = typecheckPlugin
  , spliceRunAction = metaPlugin
  , interfaceLoadAction = interfaceLoadPlugin
  }

parsedPlugin :: [CommandLineOption] -> ModSummary
             -> ParsedResult -> Hsc ParsedResult
parsedPlugin _ _ parsed@(ParsedResult pm msgs)
     = do dflags <- getDynFlags
```

(continues on next page)

(continued from previous page)

```
          liftIO $ putStrLn $ "parsePlugin: \n" ++ (showSDoc dflags $ ppr $ hpm_
↪module pm)
          liftIO $ putStrLn $ "parsePlugin warnings: \n" ++ (showSDoc dflags $ ppr $␣
↪psWarnings msgs)
          liftIO $ putStrLn $ "parsePlugin errors: \n" ++ (showSDoc dflags $ ppr $␣
↪psErrors msgs)
          return parsed

renamedAction :: [CommandLineOption] -> TcGblEnv -> HsGroup GhcRn -> TcM (TcGblEnv,␣
↪HsGroup GhcRn)
renamedAction _ tc gr = do
  dflags <- getDynFlags
  liftIO $ putStrLn $ "typeCheckPlugin (rn): " ++ (showSDoc dflags $ ppr gr)
  return (tc, gr)

typecheckPlugin :: [CommandLineOption] -> ModSummary -> TcGblEnv -> TcM TcGblEnv
typecheckPlugin _ _ tc
  = do dflags <- getDynFlags
       liftIO $ putStrLn $ "typeCheckPlugin (rn): \n" ++ (showSDoc dflags $ ppr $ tcg_
↪rn_decls tc)
       liftIO $ putStrLn $ "typeCheckPlugin (tc): \n" ++ (showSDoc dflags $ ppr $ tcg_
↪binds tc)
       return tc

metaPlugin :: [CommandLineOption] -> LHsExpr GhcTc -> TcM (LHsExpr GhcTc)
metaPlugin _ meta
  = do dflags <- getDynFlags
       liftIO $ putStrLn $ "meta: " ++ (showSDoc dflags $ ppr meta)
       return meta

interfaceLoadPlugin :: [CommandLineOption] -> ModIface -> IfM lcl ModIface
interfaceLoadPlugin _ iface
  = do dflags <- getDynFlags
       liftIO $ putStrLn $ "interface loaded: " ++ (showSDoc dflags $ ppr $ mi_module␣
↪iface)
       return iface
```

When you compile a simple module that contains Template Haskell splice

```
{-# OPTIONS_GHC -fplugin SourcePlugin #-}
{-# LANGUAGE TemplateHaskell #-}
module A where

a = ()

$(return [])
```

with the compiler flags `-fplugin SourcePlugin` it will give the following output:

```
parsePlugin:
module A where
a = ()
$(return [])
parsePlugin warnings:

parsePlugin errors:
```

(continues on next page)

(continued from previous page)

```
typeCheckPlugin (rn): a = ()
interface loaded: Language.Haskell.TH.Lib.Internal
meta: return []
typeCheckPlugin (rn):
typeCheckPlugin (rn):
Nothing
typeCheckPlugin (tc):
{$trModule = Module (TrNameS "main"#) (TrNameS "A"#), a = ()}
```

7.3.6 Hole fit plugins

Hole-fit plugins are plugins that are called when a typed-hole error message is being generated, and allows you to access information about the typed-hole at compile time, and allows you to customize valid hole fit suggestions.

Using hole-fit plugins, you can extend the behavior of valid hole fit suggestions to use e.g. Hoogle or other external tools to find and/or synthesize valid hole fits, with the same information about the typed-hole that GHC uses.

There are two access points are bundled together for defining hole fit plugins, namely a candidate plugin and a fit plugin, for modifying the candidates to be checked and fits respectively.

```
type CandPlugin = TypedHole -> [HoleFitCandidate] -> TcM [HoleFitCandidate]

type FitPlugin =  TypedHole -> [HoleFit] -> TcM [HoleFit]

data HoleFitPlugin = HoleFitPlugin
  { candPlugin :: CandPlugin
      -- ^ A plugin for modifying hole fit candidates before they're checked
  , fitPlugin :: FitPlugin
      -- ^ A plugin for modifying valid hole fits after they've been found.
  }
```

Where `TypedHole` contains all the information about the hole available to GHC at error generation.

```
data TypedHole = TyH { tyHRelevantCts :: Cts
                       -- ^ Any relevant Cts to the hole
                     , tyHImplics :: [Implication]
                       -- ^ The nested implications of the hole with the
                       --   innermost implication first.
                     , tyHCt :: Maybe Ct
                       -- ^ The hole constraint itself, if available.
                     }
```

`HoleFitPlugins` are then defined as follows

```
plugin :: Plugin
plugin = defaultPlugin {
    holeFitPlugin = (fmap . fmap) fromPureHFPlugin hfPlugin
  }


hfPlugin :: [CommandLineOption] -> Maybe HoleFitPlugin
```

Where `fromPureHFPlugin :: HoleFitPlugin -> HoleFitPluginR` is a convenience function provided in the `GHC.Tc.Errors.Hole` module, for defining plugins that do not require internal state.

7.3.6.1 Stateful hole fit plugins

`HoleFitPlugins` are wrapped in a `HoleFitPluginR`, which provides a `TcRef` for the plugin to use to track internal state, and to facilitate communication between the candidate and fit plugin.

```
-- | HoleFitPluginR adds a TcRef to hole fit plugins so that plugins can
-- track internal state. Note the existential quantification, ensuring that
-- the state cannot be modified from outside the plugin.
data HoleFitPluginR = forall s. HoleFitPluginR
  { hfPluginInit :: TcM (TcRef s)
    -- ^ Initializes the TcRef to be passed to the plugin
  , hfPluginRun :: TcRef s -> HoleFitPlugin
    -- ^ The function defining the plugin itself
  , hfPluginStop :: TcRef s -> TcM ()
    -- ^ Cleanup of state, guaranteed to be called even on error
  }
```

The plugin is then defined as by providing a value for the `holeFitPlugin` field, a function that takes the `CommandLineOption` strings that are passed to the compiler using the *-fplugin-opt=⟨module⟩:⟨args⟩* (page 625) flags and returns a `HoleFitPluginR`. This function can be used to pass the `CommandLineOption` strings along to the candidate and fit plugins respectively.

7.3.6.2 Hole fit plugin example

The following plugins allows users to limit the search for valid hole fits to certain modules, to sort the hole fits by where they originated (in ascending or descending order), as well as allowing users to put a limit on how much time is spent on searching for valid hole fits, after which new searches are aborted.

```
{-# LANGUAGE TypeApplications, RecordWildCards #-}
module HolePlugin where

import GHC.Plugins hiding ((<>))

import GHC.Tc.Errors.Hole

import Data.List (stripPrefix, sortOn)

import GHC.Tc.Types

import GHC.Tc.Utils.Monad

import Data.Time (UTCTime, NominalDiffTime)
import qualified Data.Time as Time

import Text.Read


data HolePluginState = HPS { timeAlloted :: Maybe NominalDiffTime
```

(continues on next page)

(continued from previous page)

```
                            , elapsedTime :: NominalDiffTime
                            , timeCurStarted :: UTCTime }

bumpElapsed :: NominalDiffTime -> HolePluginState -> HolePluginState
bumpElapsed ad (HPS a e t) = HPS a (e + ad) t

setAlloted :: Maybe NominalDiffTime -> HolePluginState -> HolePluginState
setAlloted a (HPS _ e t) = HPS a e t

setCurStarted :: UTCTime -> HolePluginState -> HolePluginState
setCurStarted nt (HPS a e _) = HPS a e nt

hpStartState :: HolePluginState
hpStartState = HPS Nothing zero undefined
  where zero = fromInteger @NominalDiffTime 0

initPlugin :: [CommandLineOption] -> TcM (TcRef HolePluginState)
initPlugin [msecs] = newTcRef $ hpStartState { timeAlloted = alloted }
  where
    errMsg = "Invalid amount of milliseconds given to plugin: " <> show msecs
    alloted = case readMaybe @Integer msecs of
      Just millisecs -> Just $ fromInteger @NominalDiffTime millisecs / 1000
      _ -> error errMsg
initPlugin _ = newTcRef hpStartState

fromModule :: HoleFitCandidate -> [String]
fromModule (GreHFCand gre) =
  map (moduleNameString . importSpecModule) $ gre_imp gre
fromModule _ = []

toHoleFitCommand :: TypedHole -> String -> Maybe String
toHoleFitCommand TyH{tyHCt = Just (CHoleCan _ h)} str
    = stripPrefix ("_" <> str) $ occNameString $ holeOcc h
toHoleFitCommand _ _ = Nothing

-- | This candidate plugin filters the candidates by module,
-- using the name of the hole as module to search in
modFilterTimeoutP :: [CommandLineOption] -> TcRef HolePluginState -> CandPlugin
modFilterTimeoutP _ ref hole cands = do
  curTime <- liftIO Time.getCurrentTime
  HPS {..} <- readTcRef ref
  updTcRef ref (setCurStarted curTime)
  return $ case timeAlloted of
    -- If we're out of time we remove all the candidates. Then nothing is checked.
    Just sofar | elapsedTime > sofar -> []
    _ -> case toHoleFitCommand hole "only_" of

          Just modName -> filter (inScopeVia modName) cands
          _ -> cands
  where inScopeVia modNameStr cand@(GreHFCand _) =
          elem (toModName modNameStr) $ fromModule cand
        inScopeVia _ _ = False
        toModName = replace '_' '.'
        replace :: Eq a => a -> a -> [a] -> [a]
        replace _ _ [] = []
        replace a b (x:xs) = (if x == a then b else x):replace a b xs
```

(continues on next page)

(continued from previous page)

```
modSortP :: [CommandLineOption] -> TcRef HolePluginState -> FitPlugin
modSortP _ ref hole hfs = do
  curTime <- liftIO Time.getCurrentTime
  HPS {..} <- readTcRef ref
  updTcRef ref $ bumpElapsed (Time.diffUTCTime curTime timeCurStarted)
  return $ case timeAlloted of
    -- If we're out of time, remove any candidates, so nothing is checked.
    Just sofar | elapsedTime > sofar -> [RawHoleFit $ text msg]
    _ -> case toHoleFitCommand hole "sort_by_mod" of
            -- If only_ is on, the fits will all be from the same module.
            Just ('_':'d':'e':'s':'c':_) -> reverse hfs
            Just _ -> orderByModule hfs
            _ ->  hfs
  where orderByModule :: [HoleFit] -> [HoleFit]
        orderByModule = sortOn (fmap fromModule . mbHFCand)
        mbHFCand :: HoleFit -> Maybe HoleFitCandidate
        mbHFCand HoleFit {hfCand = c} = Just c
        mbHFCand _ = Nothing
        msg = hang (text "Error: The time ran out, and the search was aborted for␣
↪this hole.")
               7 $ text "Try again with a longer timeout."

plugin :: Plugin
plugin = defaultPlugin { holeFitPlugin = holeFitP, pluginRecompile = purePlugin}

holeFitP :: [CommandLineOption] -> Maybe HoleFitPluginR
holeFitP opts = Just (HoleFitPluginR initP pluginDef stopP)
  where initP = initPlugin opts
        stopP = const $ return ()
        pluginDef ref = HoleFitPlugin { candPlugin = modFilterTimeoutP opts ref
                                      , fitPlugin  = modSortP opts ref }
```

When you then compile a module containing the following

```
{-# OPTIONS -fplugin=HolePlugin
            -fplugin-opt=HolePlugin:600
            -funclutter-valid-hole-fits #-}
module Main where

import Prelude hiding (head, last)

import Data.List (head, last)


f, g, h, i, j :: [Int] -> Int
f = _too_long
j = _
i = _sort_by_mod_desc
g = _only_Data_List
h = _only_Prelude

main :: IO ()
main = return ()
```

The output is as follows:

```
Main.hs:12:5: error:
    • Found hole: _too_long :: [Int] -> Int
      Or perhaps ‘_too_long’ is mis-spelled, or not in scope
    • In the expression: _too_long
      In an equation for ‘f’: f = _too_long
    • Relevant bindings include
        f :: [Int] -> Int (bound at Main.hs:12:1)
      Valid hole fits include
        Error: The time ran out, and the search was aborted for this hole.
               Try again with a longer timeout.
   |
12 | f = _too_long
   |     ^^^^^^^^^

Main.hs:13:5: error:
    • Found hole: _ :: [Int] -> Int
    • In the expression: _
      In an equation for ‘j’: j = _
    • Relevant bindings include
        j :: [Int] -> Int (bound at Main.hs:13:1)
      Valid hole fits include
        j :: [Int] -> Int
        f :: [Int] -> Int
        g :: [Int] -> Int
        h :: [Int] -> Int
        i :: [Int] -> Int
        head :: forall a. [a] -> a
        (Some hole fits suppressed; use -fmax-valid-hole-fits=N or -fno-max-valid-hole-
↪fits)
   |
13 | j = _
   |     ^

Main.hs:14:5: error:
    • Found hole: _sort_by_mod_desc :: [Int] -> Int
      Or perhaps ‘_sort_by_mod_desc’ is mis-spelled, or not in scope
    • In the expression: _sort_by_mod_desc
      In an equation for ‘i’: i = _sort_by_mod_desc
    • Relevant bindings include
        i :: [Int] -> Int (bound at Main.hs:14:1)
      Valid hole fits include
        sum :: forall (t :: * -> *) a. (Foldable t, Num a) => t a -> a
        product :: forall (t :: * -> *) a. (Foldable t, Num a) => t a -> a
        minimum :: forall (t :: * -> *) a. (Foldable t, Ord a) => t a -> a
        maximum :: forall (t :: * -> *) a. (Foldable t, Ord a) => t a -> a
        length :: forall (t :: * -> *) a. Foldable t => t a -> Int
        last :: forall a. [a] -> a
        (Some hole fits suppressed; use -fmax-valid-hole-fits=N or -fno-max-valid-hole-
↪fits)
   |
14 | i = _sort_by_mod_desc
   |     ^^^^^^^^^^^^^^^^^

Main.hs:15:5: error:
    • Found hole: _only_Data_List :: [Int] -> Int
      Or perhaps ‘_only_Data_List’ is mis-spelled, or not in scope
    • In the expression: _only_Data_List
```

(continues on next page)

(continued from previous page)

```
      In an equation for ‘g’: g = _only_Data_List
    • Relevant bindings include
        g :: [Int] -> Int (bound at Main.hs:15:1)
      Valid hole fits include
        head :: forall a. [a] -> a
        last :: forall a. [a] -> a
   |
15 | g = _only_Data_List
   |     ^^^^^^^^^^^^^^^

Main.hs:16:5: error:
    • Found hole: _only_Prelude :: [Int] -> Int
      Or perhaps ‘_only_Prelude’ is mis-spelled, or not in scope
    • In the expression: _only_Prelude
      In an equation for ‘h’: h = _only_Prelude
    • Relevant bindings include
        h :: [Int] -> Int (bound at Main.hs:16:1)
      Valid hole fits include
        length :: forall (t :: * -> *) a. Foldable t => t a -> Int
        maximum :: forall (t :: * -> *) a. (Foldable t, Ord a) => t a -> a
        minimum :: forall (t :: * -> *) a. (Foldable t, Ord a) => t a -> a
        product :: forall (t :: * -> *) a. (Foldable t, Num a) => t a -> a
        sum :: forall (t :: * -> *) a. (Foldable t, Num a) => t a -> a
   |
16 | h = _only_Prelude
   |     ^^^^^^^^^^^^^
```

7.3.7 Defaulting plugins

Defaulting plugins are called when ambiguous variables might otherwise cause errors, in the same way as the built-in defaulting mechanism.

A defaulting plugin can propose potential ways to fill ambiguous variables according to whatever criteria you would like. GHC will verify that those proposals will not lead to type errors in a context that you declare.

Defaulting plugins have a single access point in the *GHC.Tc.Types* module

```haskell
-- | A collection of candidate default types for sets of type variables.
data DefaultingProposal
  = DefaultingProposal
    { deProposals :: [[(TcTyVar, Type)]]
      -- ^ The type variable assignments to try.
    , deProposalCts :: [Ct]
      -- ^ The constraints against which defaults are checked.
  }

type FillDefaulting = WantedConstraints -> TcPluginM [DefaultingProposal]

-- | A plugin for controlling defaulting.
data DefaultingPlugin = forall s. DefaultingPlugin
  { dePluginInit :: TcPluginM s
    -- ^ Initialize plugin, when entering type-checker.
  , dePluginRun :: s -> FillDefaulting
    -- ^ Default some types
```

(continues on next page)

(continued from previous page)

```
  , dePluginStop :: s -> TcPluginM ()
   -- ^ Clean up after the plugin, when exiting the type-checker.
  }
```

The plugin has type `WantedConstraints -> [DefaultingProposal]`.

- It is given the currently unsolved constraints.
- It returns a list of independent "defaulting proposals".
- Each proposal of type `DefaultingProposal` specifies:
  - `deProposals`: specifies a list, in priority order, of sets of type variable assignments
  - `deProposalCts :: [Ct]` gives a set of constraints (always a subset of the incoming `WantedConstraints`) to use as a criterion for acceptance

After calling the plugin, GHC executes each `DefaultingProposal` in turn. To "execute" a proposal, GHC tries each of the proposed type assignments in `deProposals` in turn:

- It assigns the proposed types to the type variables, and then tries to solve `deProposalCts`
- If those constraints are completely solved by the assignment, GHC accepts the assignment and moves on to the next `DefaultingProposal`
- If not, GHC tries the next assignment in `deProposals`.

The plugin can assume that the incoming constraints are fully "zonked" (see the Wiki page on zonking).

The most robust `deProposalCts` to provide is the list of all wanted constraints that mention the variable you are defaulting. If you leave out a constraint, the default may be accepted, and then potentially result in a type checker error if it is incompatible with one of the constraints you left out. This can be a useful way of forcing a default and reporting errors to the user.

There is an example of defaulting lifted types in the GHC test suite. In the *testsuite/tests/plugins/* directory see *defaulting-plugin/* for the implementation, *test-defaulting-plugin.hs* for an example of when defaulting happens, and *test-defaulting-plugin-fail.hs* for an example of when defaults don't fit and aren't applied.

7.3.8 Controlling Recompilation

By default, modules compiled with plugins are always recompiled even if the source file is unchanged. This most conservative option is taken due to the ability of plugins to perform arbitrary IO actions. In order to control the recompilation behaviour you can modify the `pluginRecompile` field in `Plugin`.

```
plugin :: Plugin
plugin = defaultPlugin {
  installCoreToDos = install,
  pluginRecompile = purePlugin
  }
```

By inspecting the example `plugin` defined above, we can see that it is pure. This means that if the two modules have the same fingerprint then the plugin will always return the same result. Declaring a plugin as pure means that the plugin will never cause a module to be recompiled.

In general, the `pluginRecompile` field has the following type:

```
pluginRecompile :: [CommandLineOption] -> IO PluginRecompile
```

The PluginRecompile data type is an enumeration determining how the plugin should affect recompilation.

```
data PluginRecompile = ForceRecompile | NoForceRecompile | MaybeRecompile Fingerprint
```

A plugin which declares itself impure using ForceRecompile will always trigger a recompilation of the current module. NoForceRecompile is used for "pure" plugins which don't need to be rerun unless a module would ordinarily be recompiled. MaybeRecompile computes a Fingerprint and if this Fingerprint is different to a previously computed Fingerprint for the plugin, then we recompile the module.

As such, purePlugin is defined as a function which always returns NoForceRecompile.

```
purePlugin :: [CommandLineOption] -> IO PluginRecompile
purePlugin _ = return NoForceRecompile
```

Users can use the same functions that GHC uses internally to compute fingerprints. The GHC.Fingerprint module provides useful functions for constructing fingerprints. For example, combining together fingerprintFingerprints and fingerprintString provides an easy to to naively fingerprint the arguments to a plugin.

```
pluginFlagRecompile :: [CommandLineOption] -> IO PluginRecompile
pluginFlagRecompile =
  return . MaybeRecompile . fingerprintFingerprints . map fingerprintString . sort
```

defaultPlugin defines pluginRecompile to be impurePlugin which is the most conservative and backwards compatible option.

```
impurePlugin :: [CommandLineOption] -> IO PluginRecompile
impurePlugin _ = return ForceRecompile
```

7.3.9 Frontend plugins

A frontend plugin allows you to add new major modes to GHC. You may prefer this over a traditional program which calls the GHC API, as GHC manages a lot of parsing flags and administrative nonsense which can be difficult to manage manually. To load a frontend plugin exported by Foo.FrontendPlugin, we just invoke GHC with the *--frontend ⟨module⟩* (page 69) flag as follows:

```
$ ghc --frontend Foo.FrontendPlugin ...other options...
```

Frontend plugins, like compiler plugins, are exported by registered plugins. However, unlike compiler modules, frontend plugins are modules that export at least a single identifier frontendPlugin of type GHC.Plugins.FrontendPlugin.

FrontendPlugin exports a field frontend, which is a function [String] -> [(String, Maybe Phase)] -> Ghc (). The first argument is a list of extra flags passed to the frontend with -ffrontend-opt; the second argument is the list of arguments, usually source files and module names to be compiled (the Phase indicates if an -x flag was set), and a frontend simply executes some operation in the Ghc monad (which, among other things, has a Session).

As a quick example, here is a frontend plugin that prints the arguments that were passed to it, and then exits.

```haskell
module DoNothing.FrontendPlugin (frontendPlugin) where
import GHC.Plugins

frontendPlugin :: FrontendPlugin
frontendPlugin = defaultFrontendPlugin {
  frontend = doNothing
  }

doNothing :: [String] -> [(String, Maybe Phase)] -> Ghc ()
doNothing flags args = do
    liftIO $ print flags
    liftIO $ print args
```

Provided you have compiled this plugin and registered it in a package, you can just use it by specifying `--frontend DoNothing.FrontendPlugin` on the command line to GHC.

7.3.10 DynFlags plugins

A DynFlags plugin allows you to modify the `DynFlags` that GHC is going to use when processing a given (set of) file(s). `DynFlags` is a record containing all sorts of configuration and command line data, from verbosity level to the integer library to use, including compiler hooks, plugins and pretty-printing options. DynFlags plugins allow plugin authors to update any of those values before GHC starts doing any actual work, effectively meaning that the updates specified by the plugin will be taken into account and influence GHC's behaviour.

One of the motivating examples was the ability to register compiler hooks from a plugin. For example, one might want to modify the way Template Haskell code is executed. This is achievable by updating the `hooks` field of the `DynFlags` type, recording our custom "meta hook" in the right place. A simple application of this idea can be seen below:

```haskell
module DynFlagsPlugin (plugin) where

import BasicTypes
import GHC.Plugins
import GHC.Hs.Expr
import Language.Haskell.Syntax.Extension
import GHC.Hs.Lit
import Hooks
import GHC.Tc.Utils.Monad

plugin :: Plugin
plugin = driverPlugin { driverPlugin = hooksP }

hooksP :: [CommandLineOption] -> HscEnv -> IO HscEnv
hooksP opts hsc_env = do
    let hooks'   = (hsc_hooks hsc_env)
                     { runMetaHook = Just (fakeRunMeta opts) }
        hsc_env' = hsc_env { hsc_hooks = hooks' }
    return hsc_env'

-- This meta hook doesn't actually care running code in splices,
-- it just replaces any expression splice with the "0"
-- integer literal, and errors out on all other types of
-- meta requests.
fakeRunMeta :: [CommandLineOption] -> MetaHook TcM
```

(continues on next page)

(continued from previous page)

```
fakeRunMeta opts (MetaE r) _ = do
  liftIO . putStrLn $ "Options = " ++ show opts
  pure $ r zero

  where zero :: LHsExpr GhcPs
        zero = L noSrcSpan $ HsLit NoExtField $
          HsInt NoExtField (mkIntegralLit (0 :: Int))

fakeRunMeta _ _ _ = error "fakeRunMeta: unimplemented"
```

This simple plugin takes over the execution of Template Haskell code, replacing any expression splice it encounters by `0` (at type `Int`), and errors out on any other type of splice.

Therefore, if we run GHC against the following code using the plugin from above:

```
{-# OPTIONS -fplugin=DynFlagsPlugin #-}
{-# LANGUAGE TemplateHaskell #-}
module Main where

main :: IO ()
main = print $( [|1|] )
```

This will not actually evaluate `[|1|]`, but instead replace it with the `0 :: Int` literal.

Just like the other types of plugins, you can write `DynFlags` plugins that can take and make use of some options that you can then specify using the `-fplugin-opt` flag. In the `DynFlagsPlugin` code from above, the said options would be available in the `opts` argument of `hooksP`.

Finally, since those `DynFlags` updates happen after the plugins are loaded, you cannot from a `DynFlags` plugin register other plugins by just adding them to the `plugins` field of `DynFlags`. In order to achieve this, you would have to load them yourself and store the result into the `cachedPlugins` field of `DynFlags`.

7.4 Referring to back ends

In versions of GHC numbered up to and including 9.4, a back end is referred to by name: type `Backend`, from module `GHC.Driver.Backend`, is a simple enumeration type. In versions of GHC numbered 9.6 and higher, `Backend` is an abstract type. The module specifies predicates and functions associated with a back end.

This change in representation requires changes in client code.

7.4.1 Client code that only names back ends

Suppose your client uses `Backend` only to mention back ends by name. That is, it never discriminates between back ends in a `case` expression, function definition, or equality comparison. Then the simplest way for you to migrate your code is to replace each value constructor from version 9.4 with the corresponding value from 9.6:

Old value	New value
NCG	ncgBackend
LLVM	llvmBackend
ViaC	viaCBackend
Interpreter	interpreterBackend
NoBackend	noBackend

7.4.2 Client code that discriminates among back ends

Suppose your code makes decisions based on the value of an expression of type `Backend`. Then the simplest way for you to migrate your decision-making code depends on the code's form.

- If your decision-making is driven by an equality or inequality predicate, an equivalent predicate may already be defined in module `GHC.Driver.Backend`. For example, if your client wants to be sure that optimization levels above `-O0` are permitted, it might have originally compared `backend /= Interpreter`. But now there is a predicate for that: it is `not (backendForcesOptimization0 backend)`.

  If the predicate you want is not already defined, you will have to fall back on the more general strategy defined below.

- If your decision-making is still driven by a predicate, but the implementation of the predicate inspects the form of `Backend`, you may still be in luck. For example, if your client needs to know whether the `Backend` wishes to write files to disk, it can query `backendWritesFiles backend`. In version 9.4, this predicate holds for the NCG, LLVM, and Via-C back ends, but not for the interpreter or for `NoBackend`.

- In the general case, for any function definition, case expression, or equality test that discriminates among back ends, you can use the general migration strategy described below.

7.4.3 General migration strategy for client code

From version 9.6 onward, each back end may be queried for its name:

```
backendName :: Backend -> BackendName
```

The `BackendName` type must be imported from module `GHC.Driver.Backend.Internal`. It is defined to look the same as the old `Backend` type:

```
data BackendName
   = NCG
   | LLVM
   | ViaC
   | Interpreter
   | NoBackend
```

This type is also an instance of the `Eq` and `Show` classes.

If your existing code discriminates among existing back ends using a `case` expression, you need to apply `backendName` to the scrutinee.

```
case backend dflags of  -- code using the 9.4 interface
  NCG -> ...
  LLVM -> ...
  ...
```

can become

```
case backendName $ backend dflags of  -- code using the 9.6 interface
  NCG -> ...
  LLVM -> ...
  ...
```

Only the scrutinee changes, not the pattern matches. And if your pattern matches were complete before, they are still complete.

CHAPTER

EIGHT

PROFILING

GHC comes with a time and space profiling system, so that you can answer questions like "why is my program so slow?", or "why is my program using so much memory?". We'll start by describing how to do time profiling.

Time profiling a program is a three-step process:

1. Re-compile your program for profiling with the *-prof* (page 656) option, and probably one of the options for adding automatic annotations: *-fprof-late* (page 657) is the recommended option.
2. Having compiled the program for profiling, you now need to run it to generate the profile. For example, a simple time profile can be generated by running the program with `+RTS -p` (see *-p* (page 659)), which generates a file named *prog*`.prof` where ⟨prog⟩ is the name of your program (without the `.exe` extension, if you are on Windows).

   There are many different kinds of profile that can be generated, selected by different RTS options. We will be describing the various kinds of profile throughout the rest of this chapter. Some profiles require further processing using additional tools after running the program.
3. Examine the generated profiling information, use the information to optimise your program, and repeat as necessary.

The time profiler measures the CPU time taken by the Haskell code in your application. In particular time taken by safe foreign calls is not tracked by the profiler (see *Profiling and foreign calls* (page 656)).

8.1 Cost centres and cost-centre stacks

GHC's profiling system assigns costs to cost centres. A cost is simply the time or space (memory) required to evaluate an expression. Cost centres are program annotations around expressions; all costs incurred by the annotated expression are assigned to the enclosing cost centre. Furthermore, GHC will remember the stack of enclosing cost centres for any given expression at run-time and generate a call-tree of cost attributions.

Let's take a look at an example:

```
main = print (fib 30)
fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

Compile and run this program as follows:

```
$ ghc -prof -fprof-auto -rtsopts Main.hs
$ ./Main +RTS -p
121393
$
```

When a GHC-compiled program is run with the *-p* (page 659) RTS option, it generates a file called `prog.prof`. In this case, the file will contain something like this:

```
        Wed Oct 12 16:14 2011 Time and Allocation Profiling Report  (Final)

           Main +RTS -p -RTS

        total time  =        0.68 secs   (34 ticks @ 20 ms)
        total alloc = 204,677,844 bytes  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc

fib         Main    100.0  100.0


                                                      individual     inherited
COST CENTRE MODULE                  no.     entries  %time %alloc   %time %alloc

MAIN        MAIN                    102           0    0.0    0.0   100.0  100.0
 CAF        GHC.IO.Handle.FD        128           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding.Iconv   120           0    0.0    0.0     0.0    0.0
 CAF        GHC.Conc.Signal         110           0    0.0    0.0     0.0    0.0
 CAF        Main                    108           0    0.0    0.0   100.0  100.0
  main      Main                    204           1    0.0    0.0   100.0  100.0
   fib      Main                    205     2692537  100.0  100.0   100.0  100.0
```

The first part of the file gives the program name and options, and the total time and total memory allocation measured during the run of the program (note that the total memory allocation figure isn't the same as the amount of *live* memory needed by the program at any one time; the latter can be determined using heap profiling, which we will describe later in *Profiling memory usage* (page 663)).

The second part of the file is a break-down by cost centre of the most costly functions in the program. In this case, there was only one significant function in the program, namely `fib`, and it was responsible for 100% of both the time and allocation costs of the program.

The third and final section of the file gives a profile break-down by cost-centre stack. This is roughly a call-tree profile of the program. In the example above, it is clear that the costly call to `fib` came from `main`.

The time and allocation incurred by a given part of the program is displayed in two ways: "individual", which are the costs incurred by the code covered by this cost centre stack alone, and "inherited", which includes the costs incurred by all the children of this node.

The usefulness of cost-centre stacks is better demonstrated by modifying the example slightly:

```haskell
main = print (f 30 + g 30)
  where
    f n  = fib n
    g n  = fib (n `div` 2)

fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

Compile and run this program as before, and take a look at the new profiling results:

```
COST CENTRE MODULE                    no.     entries  %time %alloc   %time %alloc

MAIN        MAIN                      102           0    0.0    0.0   100.0  100.0
 CAF        GHC.IO.Handle.FD          128           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding.Iconv     120           0    0.0    0.0     0.0    0.0
 CAF        GHC.Conc.Signal           110           0    0.0    0.0     0.0    0.0
 CAF        Main                      108           0    0.0    0.0   100.0  100.0
  main      Main                      204           1    0.0    0.0   100.0  100.0
   main.g   Main                      207           1    0.0    0.0     0.0    0.1
    fib     Main                      208        1973    0.0    0.1     0.0    0.1
   main.f   Main                      205           1    0.0    0.0   100.0   99.9
    fib     Main                      206     2692537  100.0   99.9   100.0   99.9
```

Now although we had two calls to `fib` in the program, it is immediately clear that it was the call from `f` which took all the time. The functions `f` and `g` which are defined in the `where` clause in `main` are given their own cost centres, `main.f` and `main.g` respectively.

The actual meaning of the various columns in the output is:

The number of times this particular point in the call tree was entered.

The percentage of the total run time of the program spent at this point in the call tree.

The percentage of the total memory allocations (excluding profiling overheads) of the program made by this call.

The percentage of the total run time of the program spent below this point in the call tree.

The percentage of the total memory allocations (excluding profiling overheads) of the program made by this call and all of its sub-calls.

In addition you can use the *-P* (page 659) RTS option to get the following additional information:

ticks The raw number of time "ticks" which were attributed to this cost-centre; from this, we get the `%time` figure mentioned above.

bytes Number of bytes allocated in the heap while in this cost-centre; again, this is the raw number from which we get the `%alloc` figure mentioned above.

What about recursive functions, and mutually recursive groups of functions? Where are the costs attributed? Well, although GHC does keep information about which groups of functions called each other recursively, this information isn't displayed in the basic time and allocation profile, instead the call-graph is flattened into a tree as follows: a call to a function that occurs elsewhere on the current stack does not push another entry on the stack, instead the costs for this call are aggregated into the caller[2].

[2] Note that this policy has changed slightly in GHC 7.4.1 relative to earlier versions, and may yet change further, feedback is welcome.

8.1.1 Inserting cost centres by hand

Cost centres are just program annotations. When you say `-fprof-auto` to the compiler, it automatically inserts a cost centre annotation around every binding not marked INLINE in your program, but you are entirely free to add cost centre annotations yourself. Be careful adding too many cost-centre annotations as the optimiser is careful to not move them around or remove them, which can severly affect how your program is optimised and hence the runtime performance!

The syntax of a cost centre annotation for expressions is

```
{-# SCC "name" #-} <expression>
```

where `"name"` is an arbitrary string, that will become the name of your cost centre as it appears in the profiling output, and `<expression>` is any Haskell expression. An `SCC` annotation extends as far to the right as possible when parsing, having the same precedence as lambda abstractions, let expressions, and conditionals. Additionally, an annotation may not appear in a position where it would change the grouping of subexpressions:

```
a = 1 / 2 / 2                              -- accepted (a=0.25)
b = 1 / {-# SCC "name" #-} 2 / 2           -- rejected (instead of b=1.0)
```

This restriction is required to maintain the property that inserting a pragma, just like inserting a comment, does not have unintended effects on the semantics of the program, in accordance with GHC Proposal #176.

SCC stands for "Set Cost Centre". The double quotes can be omitted if `name` is a Haskell identifier starting with a lowercase letter, for example:

```
{-# SCC id #-} <expression>
```

Cost centre annotations can also appear in the top-level or in a declaration context. In that case you need to pass a function name defined in the same module or scope with the annotation. Example:

```
f x y = ...
  where
    g z = ...
    {-# SCC g #-}

{-# SCC f #-}
```

If you want to give a cost centre different name than the function name, you can pass a string to the annotation

```
f x y = ...
{-# SCC f "cost_centre_name" #-}
```

Here is an example of a program with a couple of SCCs:

```
main :: IO ()
main = do let xs = [1..1000000]
          let ys = [1..2000000]
          print $ {-# SCC last_xs #-} last xs
          print $ {-# SCC last_init_xs #-} last (init xs)
          print $ {-# SCC last_ys #-} last ys
          print $ {-# SCC last_init_ys #-} last (init ys)
```

which gives this profile when run:

```
COST CENTRE     MODULE                  no.     entries  %time %alloc   %time %alloc

MAIN            MAIN                    102           0    0.0    0.0   100.0  100.0
 CAF            GHC.IO.Handle.FD        130           0    0.0    0.0     0.0    0.0
 CAF            GHC.IO.Encoding.Iconv   122           0    0.0    0.0     0.0    0.0
 CAF            GHC.Conc.Signal         111           0    0.0    0.0     0.0    0.0
 CAF            Main                    108           0    0.0    0.0   100.0  100.0
  main          Main                    204           1    0.0    0.0   100.0  100.0
   last_init_ys Main                    210           1   25.0   27.4    25.0   27.4
   main.ys      Main                    209           1   25.0   39.2    25.0   39.2
   last_ys      Main                    208           1   12.5    0.0    12.5    0.0
   last_init_xs Main                    207           1   12.5   13.7    12.5   13.7
   main.xs      Main                    206           1   18.8   19.6    18.8   19.6
   last_xs      Main                    205           1    6.2    0.0     6.2    0.0
```

8.1.2 Rules for attributing costs

While running a program with profiling turned on, GHC maintains a cost-centre stack behind the scenes, and attributes any costs (memory allocation and time) to whatever the current cost-centre stack is at the time the cost is incurred.

The mechanism is simple: whenever the program evaluates an expression with an SCC annotation, `{-# SCC c -#} E`, the cost centre `c` is pushed on the current stack, and the entry count for this stack is incremented by one. The stack also sometimes has to be saved and restored; in particular when the program creates a thunk (a lazy suspension), the current cost-centre stack is stored in the thunk, and restored when the thunk is evaluated. In this way, the cost-centre stack is independent of the actual evaluation order used by GHC at runtime.

At a function call, GHC takes the stack stored in the function being called (which for a top-level function will be empty), and *appends* it to the current stack, ignoring any prefix that is identical to a prefix of the current stack.

We mentioned earlier that lazy computations, i.e. thunks, capture the current stack when they are created, and restore this stack when they are evaluated. What about top-level thunks? They are "created" when the program is compiled, so what stack should we give them? The technical name for a top-level thunk is a CAF ("Constant Applicative Form"). GHC assigns every CAF in a module a stack consisting of the single cost centre `M.CAF`, where `M` is the name of the module. It is also possible to give each CAF a different stack, using the option *`-fprof-cafs`* (page 658). This is especially useful when compiling with *`-ffull-laziness`* (page 118) (as is default with *`-O`* (page 113) and higher), as constants in function bodies will be lifted to the top-level and become CAFs. You will probably need to consult the Core (*`-ddump-simpl`* (page 259)) in order to determine what these CAFs correspond to.

8.2 Profiling and foreign calls

Simply put, the profiler includes time spent in unsafe foreign calls but ignores time taken in safe foreign calls. For example, time spent blocked on IO operations (e.g. `getLine`) is not accounted for in the profile as `getLine` is implemented using a safe foreign call.

The profiler estimates CPU time, for Haskell threads within the program only. In particular, time "taken" by the program in blocking safe foreign calls is not accounted for in time profiles. The runtime has the notion of a virtual processor which is known as a "capability". Haskell threads are run on capabilities, and the profiler samples the capabilities in order to determine what is being executed at a certain time. When a safe foreign call is executed, it's run outside the context of a capability; hence the sampling does not account for the time taken. Whilst the safe call is executed, other Haskell threads are free to run on the capability, and their cost will be attributed to the profiler. When the safe call is finished, the blocked, descheduled thread can be resumed and rescheduled.

However, the time taken by blocking on unsafe foreign calls is accounted for in the profile. This happens because unsafe foreign calls are executed by the same capability their calling Haskell thread is running on. Therefore, an unsafe foreign call will block the entire capability whilst it is running, and any time the capability is sampled the "cost" of the foreign call will be attributed to the calling cost-centre stack.

However, do note that you are not supposed to use unsafe foreign calls for any operations which do block! Do not be tempted to replace your safe foreign calls with unsafe calls just so they appear in the profile. This prevents GC from happening until the foreign call returns, which can be catastrophic for performance.

8.3 Compiler options for profiling

`-prof`

To make use of the profiling system *all* modules must be compiled and linked with the *`-prof`* (page 656) option. Any `SCC` annotations you've put in your source will spring to life.

Without a *`-prof`* (page 656) option, your `SCC`s are ignored; so you can compile `SCC`-laden code without changing it.

`-fno-prof-count-entries`

Tells GHC not to collect information about how often functions are entered at runtime (the "entries" column of the time profile), for this module. This tends to make the profiled code run faster, and hence closer to the speed of the unprofiled code, because GHC is able to optimise more aggressively if it doesn't have to maintain correct entry counts. This option can be useful if you aren't interested in the entry counts (for example, if you only intend to do heap profiling).

There are a few other profiling-related compilation options. Use them *in addition to* *`-prof`* (page 656). These do not have to be used consistently for all modules in a program.

8.3.1 Automatically placing cost-centres

GHC has a number of flags for automatically inserting cost-centres into the compiled program. Use these options carefully because inserting too many cost-centres in the wrong places will mean the optimiser will be less effective and the runtime behaviour of your profiled program will be different to that of the unprofiled one.

-fprof-callers=⟨name⟩

Automatically enclose all occurrences of the named function in an `SCC`. Note that these cost-centres are added late in compilation (after simplification) and consequently the names may be slightly different than they appear in the source program (e.g. a call to `f` may inlined with its wrapper, resulting in an occurrence of its worker, `$wf`).

In addition to plain module-qualified names (e.g. `GHC.Base.map`), ⟨name⟩ also accepts a small globbing language using `*` as a wildcard symbol:

```
pattern    := <module> '.' <identifier>
module     := '*'
            | <Haskell module name>
identifier := <ident_char>
ident
```

For instance, the following are all valid patterns:

- `Data.List.map`
- `*.map`
- `*.parse*`
- `*.<\*>`

The `*` character can be used literally by escaping (e.g. `\*`).

-fprof-auto

All bindings not marked *INLINE* (page 609), whether exported or not, top level or nested, will be given automatic `SCC` annotations. Functions marked *INLINE* (page 609) must be given a cost centre manually.

-fprof-auto-top

GHC will automatically add `SCC` annotations for all top-level bindings not marked *INLINE* (page 609). If you want a cost centre on an *INLINE* (page 609) function, you have to add it manually.

-fprof-auto-exported

GHC will automatically add `SCC` annotations for all exported functions not marked *INLINE* (page 609). If you want a cost centre on an *INLINE* (page 609) function, you have to add it manually.

-fprof-auto-calls

Adds an automatic `SCC` annotation to all *call sites*. This is particularly useful when using profiling for the purposes of generating stack traces; see the function Debug.Trace.traceShow, or the *-xc* (page 198) RTS flag (*RTS options for Haskell program coverage* (page 196)) for more details.

-fprof-late

Since 9.4.1

Adds an automatic `SCC` annotation to all top level bindings late in the compilation pipeline after the optimizer has run and unfoldings have been created. This means these cost centres will not interfere with core-level optimizations and the resulting profile will be closer to the performance profile of an optimized non-profiled executable. While the results of this are generally informative, some of the compiler internal names will leak into the profile. Further if a function is inlined into a use site it's costs will be counted against the caller's cost center.

For example if we have this code:

```
{-# INLINE mysum #-}
mysum = sum
main = print $ mysum [1..9999999]
```

Then `mysum` will not show up in the profile since it will be inlined into main and therefore it's associated costs will be attributed to mains implicit cost centre.

`-fprof-late-inline`

Since 9.4.1

Adds an automatic `SCC` annotation to all top level bindings late in the core pipeline after the optimizer has run. This is the same as *`-fprof-late`* (page 657) except that cost centers are included in some unfoldings.

The result of which is that cost centers *can* inhibit core optimizations to some degree at use sites after inlining. Further there can be significant overhead from cost centres added to small functions if they are inlined often.

You can try this mode if *`-fprof-late`* (page 657) results in a profile that's too hard to interpret.

`-fprof-late-overloaded`

Since 9.10.1

Adds an automatic `SCC` annotation to all *overloaded* top level bindings late in the compilation pipeline after the optimizer has run and unfoldings have been created. This means these cost centres will not interfere with core-level optimizations and the resulting profile will be closer to the performance profile of an optimized non-profiled executable.

This flag can help determine which top level bindings encountered during a program's execution are still overloaded after inlining and specialization.

`-fprof-late-overloaded-calls`

Since 9.10.1

Adds an automatic `SCC` annotation to all call sites that include dictionary arguments late in the compilation pipeline after the optimizer has run and unfoldings have been created. This means these cost centres will not interfere with core-level optimizations and the resulting profile will be closer to the performance profile of an optimized non-profiled executable.

This flag is potentially more useful than *`-fprof-late-overloaded`* (page 658) since it will also add `SCC` annotations to call sites of imported overloaded functions.

Some overloaded calls may not be annotated, specifically in cases where the optimizer turns an overloaded function into a join point. Calls to such functions will not be wrapped in `SCC` annotations, since it would make them non-tail calls, which is a requirement for join points. Instead, `SCC` annotations are added around the body of overloaded join variables and given distinct names (`join-rhs-<var>`) to avoid confusion.

-fprof-cafs
The costs of all CAFs in a module are usually attributed to one "big" CAF cost-centre. With this option, all CAFs get their own cost-centre. An "if all else fails" option…

-fprof-manual

Default on

Process (or ignore) manual `SCC` annotations. Can be helpful to ignore annotations from libraries which are not desired.

-auto-all
Deprecated alias for *-fprof-auto* (page 657)

-auto
Deprecated alias for *-fprof-auto-exported* (page 657)

-caf-all
Deprecated alias for *-fprof-cafs* (page 658)

-no-auto-all
Deprecated alias for *-fno-prof-auto* (page 657)

-no-auto
Deprecated alias for *-fno-prof-auto* (page 657)

-no-caf-all
Deprecated alias for *-fno-prof-cafs* (page 658)

8.4 Time and allocation profiling

To generate a time and allocation profile, give one of the following RTS options to the compiled program when you run it (RTS options should be enclosed between `+RTS ... -RTS` as usual):

-p
-P
-pa
The *-p* (page 659) option produces a standard *time profile* report. It is written into the file `<stem>.prof`; the stem is taken to be the program name by default, but can be overridden by the *-po ⟨stem⟩* (page 659) flag.

The *-P* (page 659) option produces a more detailed report containing the actual time and allocation data as well. (Not used much.)

The *-pa* (page 659) option produces the most detailed report containing all cost centres in addition to the actual time and allocation data.

-pj
The *-pj* (page 659) option produces a time/allocation profile report in JSON format written into the file `<program>.prof`.

-po ⟨stem⟩
The *-po ⟨stem⟩* (page 659) option overrides the stem used to form the output file paths for the cost-centre profiler (see *-p* (page 659) and *-pj* (page 659) flags above) and heap profiler (see *-h* (page 194)).

For instance, running a program with `+RTS -h -p -pohello-world` would produce a heap profile named `hello-world.hp` and a cost-centre profile named `hello-world.prof`.

-V ⟨secs⟩

> **Default** 0.001 when profiling, and 0.01 otherwise

Sets the interval that the RTS clock ticks at, which is also the sampling interval of the time and allocation profile. The default is 0.001 seconds when profiling, and 0.01 otherwise. The runtime uses a single timer signal to count ticks; this timer signal is used to control the context switch timer (*Using Concurrent Haskell* (page 131)) and the heap profiling timer *RTS options for heap profiling* (page 665). Also, the time profiler uses the RTS timer signal directly to record time profiling samples.

Normally, setting the *-V ⟨secs⟩* (page 659) option directly is not necessary: the resolution of the RTS timer is adjusted automatically if a short interval is requested with the *-C ⟨s⟩* (page 131) or *-i ⟨secs⟩* (page 667) options. However, setting *-V ⟨secs⟩* (page 659) is required in order to increase the resolution of the time profiler.

Using a value of zero disables the RTS clock completely, and has the effect of disabling timers that depend on it: the context switch timer and the heap profiling timer. Context switches will still happen, but deterministically and at a rate much faster than normal. Disabling the interval timer is useful for debugging, because it eliminates a source of non-determinism at runtime.

-xc

This option causes the runtime to print out the current cost-centre stack whenever an exception is raised. This can be particularly useful for debugging the location of exceptions, such as the notorious `Prelude.head: empty list` error. See *RTS options for Haskell program coverage* (page 196).

8.4.1 JSON profile format

profile in a machine-readable JSON format. The JSON file can be directly loaded into speedscope.app to interactively view the profile.

The top-level object of this format has the following properties,

program (string) The name of the program

arguments (list of strings) The command line arguments passed to the program

rts_arguments (list of strings) The command line arguments passed to the runtime system

initial_capabilities (integral number) How many capabilities the program was started with (e.g. using the *-N ⟨x⟩* (page 132) option). Note that the number of capabilities may change during execution due to the `setNumCapabilities` function.

total_time (number) The total wall time of the program's execution in seconds.

total_ticks (integral number) How many profiler "ticks" elapsed over the course of the program's execution.

end_time (number) The approximate time when the program finished execution as a UNIX epoch timestamp.

tick_interval (float) How much time between profiler ticks.

total_alloc (integer) The cumulative allocations of the program in bytes.

cost_centres (list of objects) A list of the program's cost centres

profile (object) The profile tree itself

Each entry in `cost_centres` is an object describing a cost-centre of the program having the following properties,

`id` (integral number) A unique identifier used to refer to the cost-centre

`is_caf` (boolean) Whether the cost-centre is a Constant Applicative Form (CAF)

`label` (string) A descriptive string roughly identifying the cost-centre.

`src_loc` (string) A string describing the source span enclosing the cost-centre.

The profile data itself is described by the `profile` field, which contains a tree-like object (which we'll call a "cost-centre stack" here) with the following properties,

`id` (integral number) The `id` of a cost-centre listed in the `cost_centres` list.

`entries` (integral number) How many times was this cost-centre entered?

`ticks` (integral number) How many ticks was the program's execution inside of this cost-centre? This does not include child cost-centres.

`alloc` (integral number) How many bytes did the program allocate while inside of this cost-centre? This does not include allocations while in child cost-centres.

`children` (list) A list containing child cost-centre stacks.

For instance, a simple profile might look like this,

```
{
  "program": "Main",
  "arguments": [
    "nofib/shootout/n-body/Main",
    "50000"
  ],
  "rts_arguments": [
    "-pj",
    "-hy"
  ],
  "end_time": "Thu Feb 23 17:15 2017",
  "initial_capabilities": 0,
  "total_time": 1.7,
  "total_ticks": 1700,
  "tick_interval": 1000,
  "total_alloc": 3770785728,
  "cost_centres": [
    {
      "id": 168,
      "label": "IDLE",
      "module": "IDLE",
      "src_loc": "<built-in>",
      "is_caf": false
    },
    {
      "id": 156,
      "label": "CAF",
      "module": "GHC.Integer.Logarithms.Internals",
      "src_loc": "<entire-module>",
      "is_caf": true
    },
    {
      "id": 155,
      "label": "CAF",
```

(continues on next page)

(continued from previous page)

```
      "module": "GHC.Integer.Logarithms",
      "src_loc": "<entire-module>",
      "is_caf": true
    },
    {
      "id": 154,
      "label": "CAF",
      "module": "GHC.Event.Array",
      "src_loc": "<entire-module>",
      "is_caf": true
    }
  ],
  "profile": {
    "id": 162,
    "entries": 0,
    "alloc": 688,
    "ticks": 0,
    "children": [
      {
        "id": 1,
        "entries": 0,
        "alloc": 208,
        "ticks": 0,
        "children": [
          {
            "id": 22,
            "entries": 1,
            "alloc": 80,
            "ticks": 0,
            "children": []
          }
        ]
      },
      {
        "id": 42,
        "entries": 1,
        "alloc": 1632,
        "ticks": 0,
        "children": []
      }
    ]
  }
}
```

8.4.2 Eventlog profile format

In addition to the `.prof` and `.json` formats the cost centre definitions and samples are also emitted to the *eventlog* (page 195). The format of the events is specified in the *eventlog encodings* (page 743) section.

8.5 Profiling memory usage

In addition to profiling the time and allocation behaviour of your program, you can also generate a graph of its memory usage over time. This is useful for detecting the causes of space leaks, when your program holds on to more memory at run-time that it needs to. Space leaks lead to slower execution due to heavy garbage collector activity, and may even cause the program to run out of memory altogether.

Heap profiling differs from time profiling in the fact that is not always necessary to use the profiling runtime to generate a heap profile. There are two heap profiling modes (`-hT` (page 194) and `-hi` (page 666)[1]) which are always available.

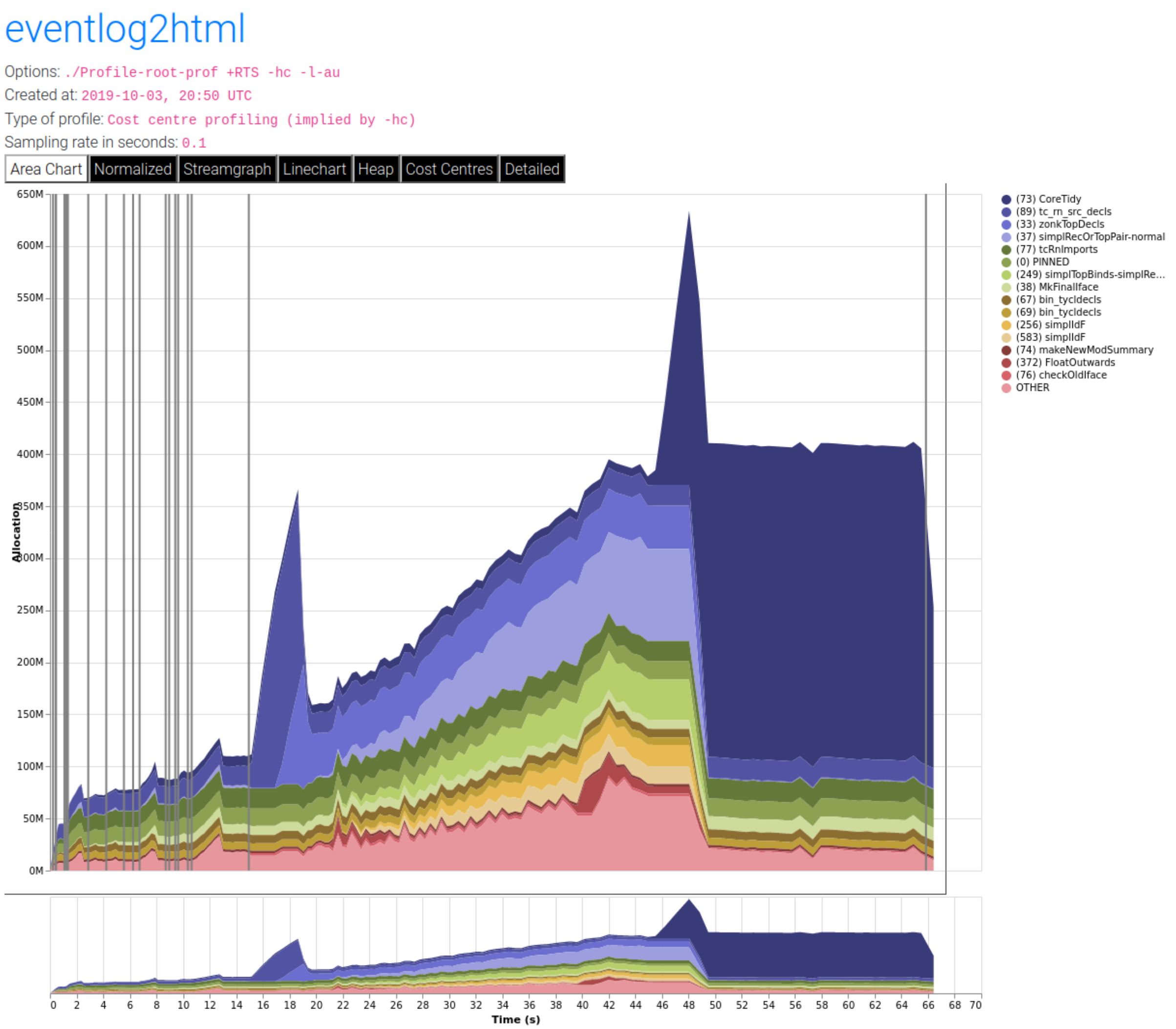
To generate a heap profile from your program:

1. Assuming you need the profiling runtime, compile the program for profiling (*Compiler options for profiling* (page 656)).
2. Run it with one of the heap profiling options described below (eg. `-hc` (page 665) for a basic producer profile) and enable the eventlog using `-l` (page 195).

   Heap samples will be emitted to the GHC event log (see *Heap profiler event log output* (page 753) for details about event format).
3. Render the heap profile using eventlog2html. This produces an HTML file which contains the visualised profile.
4. Open the rendered interactive profile in your web browser.

For example, here is a heap profile produced of using eventlog profiling on GHC compiling the Cabal library. You can read a lot more about eventlog2html on the website.

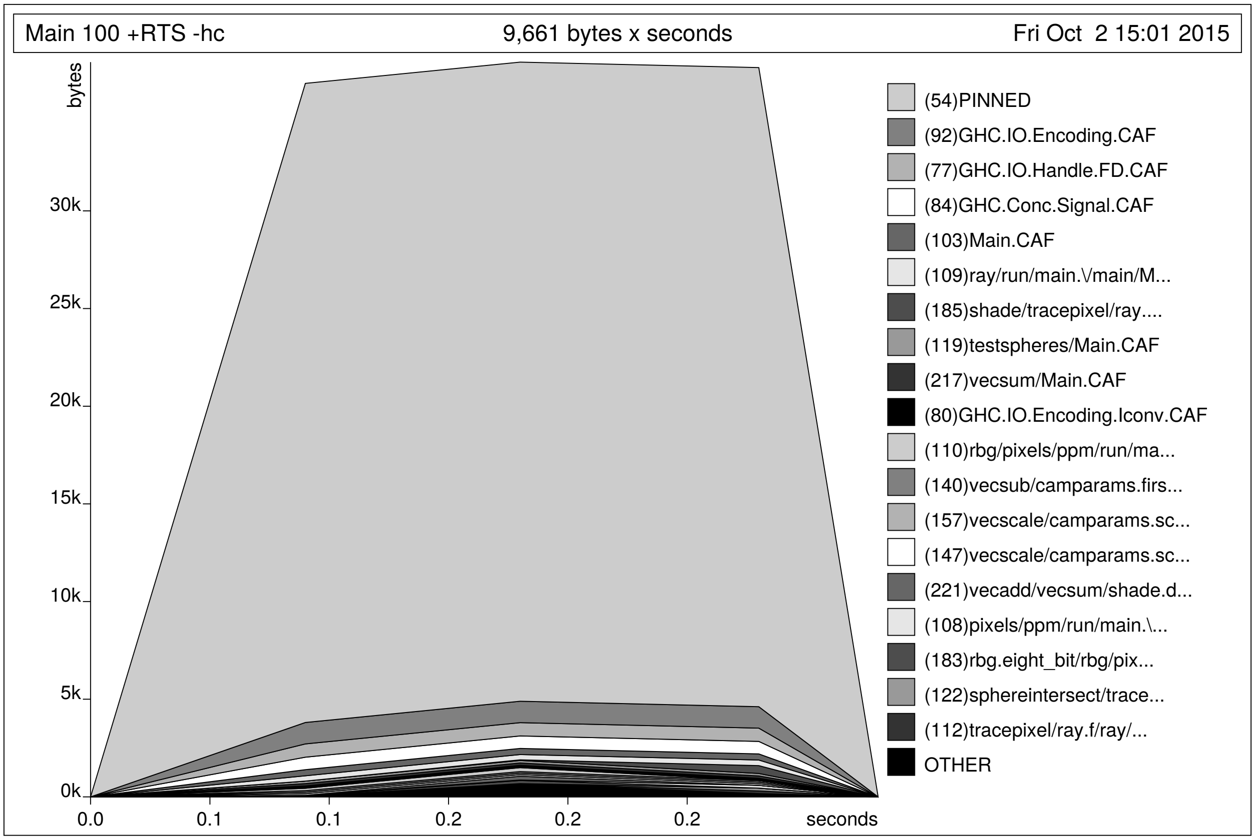
[1] `-hi` (page 666) profiling is avaible with the normal runtime but you will need to compile with `-finfo-table-map` (page 687) to interpret the results.



Note that there is the legacy *prog*.hp format which has been deprecated in favour of eventlog based profiling. In order to render the legacy format, the steps are as follows.

3. Run **hp2ps** to produce a Postscript file, *prog*.ps. The **hp2ps** utility is described in detail in *hp2ps – Rendering heap profiles to PostScript* (page 670).
4. Display the heap profile using a postscript viewer such as Ghostview, or print it out on a Postscript-capable printer.

For example, here is a heap profile produced for the `sphere` program from GHC's `nofib` benchmark suite,



Note that there might be a big difference between the OS reported memory usage of your program and the amount of live data as reported by heap profiling. The reasons for the difference are explained in *Understanding how OS memory usage corresponds to live data* (page 696).

8.5.1 RTS options for heap profiling

There are several different kinds of heap profile that can be generated. All the different profile types yield a graph of live heap against time, but they differ in how the live heap is broken down into bands. The following RTS options select which break-down to use:

-hT
: Breaks down the graph by heap closure type. This does not require the profiling runtime.

-hc
: *Requires -prof* (page 656). Breaks down the graph by the cost-centre stack which produced the data.

-hm
: *Requires -prof* (page 656). Break down the live heap by the module containing the code which produced the data.

-hd
: *Requires -prof* (page 656). Breaks down the graph by closure description. For actual data, the description is just the constructor name, for other closures it is a compiler-generated string identifying the closure.

-hy
: *Requires -prof* (page 656). Breaks down the graph by type. For closures which have

function type or unknown/polymorphic type, the string will represent an approximation to the actual type.

-he

Since 9.10.1

Requires *-prof* (page 656). Break down the graph by era.

Each closure is tagged with the era in which it is created. Eras start at 1 and can be set in your program to domain specific values using functions from `GHC.Profiling.Eras` or incremented automatically by the *--automatic-era-increment* (page 667).

-hr

Requires *-prof* (page 656). Break down the graph by retainer set. Retainer profiling is described in more detail below (*Retainer Profiling* (page 668)).

-hb

Requires *-prof* (page 656). Break down the graph by biography. Biographical profiling is described in more detail below (*Biographical Profiling* (page 669)).

-hi

Break down the graph by the address of the info table of a closure. For this to produce useful output the program must have been compiled with *-finfo-table-map* (page 687) but it does not require the profiling runtime.

-l

Emit profile samples to the *GHC event log* (page 195). This format is both more expressive than the old `.hp` format and can be correlated with other events over the program's runtime. See *Heap profiler event log output* (page 753) for details on the produced event structure.

In addition, the profile can be restricted to heap data which satisfies certain criteria - for example, you might want to display a profile by type but only for data produced by a certain module, or a profile by retainer for a certain type of data. Restrictions are specified as follows:

-hc ⟨name⟩

Requires *-prof* (page 656). Restrict the profile to closures produced by cost-centre stacks with one of the specified cost centres at the top.

-hC ⟨name⟩

Requires *-prof* (page 656). Restrict the profile to closures produced by cost-centre stacks with one of the specified cost centres anywhere in the stack.

-hm ⟨module⟩

Requires *-prof* (page 656). Restrict the profile to closures produced by the specified modules.

-hd ⟨desc⟩

Requires *-prof* (page 656). Restrict the profile to closures with the specified description strings.

-hy ⟨type⟩

Requires *-prof* (page 656). Restrict the profile to closures with the specified types.

-he ⟨era⟩

Requires *-prof* (page 656). Restrict the profile to the specified era.

-hr ⟨cc⟩

Requires *-prof* (page 656). Restrict the profile to closures with retainer sets containing cost-centre stacks with one of the specified cost centres at the top.

-hb ⟨bio⟩
Requires *-prof* (page 656). Restrict the profile to closures with one of the specified biographies, where ⟨bio⟩ is one of `lag`, `drag`, `void`, or `use`.

For example, the following options will generate a retainer profile restricted to `Branch` and `Leaf` constructors:

```
prog +RTS -hr -hdBranch,Leaf
```

There can only be one "break-down" option (eg. *-hr* (page 666) in the example above), but there is no limit on the number of further restrictions that may be applied. All the options may be combined, with one exception: GHC doesn't currently support mixing the *-hr* (page 666) and *-hb* (page 666) options.

There are three more options which relate to heap profiling:

-i ⟨secs⟩
Set the profiling (sampling) interval to ⟨secs⟩ seconds (the default is 0.1 second). Fractions are allowed: for example `-i0.2` will get 5 samples per second. This only affects heap profiling; time profiles are always sampled with the frequency of the RTS clock. See *Time and allocation profiling* (page 659) for changing that.

--no-automatic-heap-samples

Since 9.2.1

Don't start heap profiling from the start of program execution. If this option is enabled, it's expected that the user will manually start heap profiling or request specific samples using functions from `GHC.Profiling`.

--no-automatic-time-samples

Since 9.10.1

Don't start time profiling from the start of program execution. If this option is enabled, it's expected that the user will manually start time profiling or request specific samples using functions from `GHC.Profiling`.

--automatic-era-increment

Since 9.10.1

Increment the era by 1 on each major garbage collection. This is used in conjunction with *-he* (page 666).

--null-eventlog-writer

Since 9.2.2

Don't output eventlog to file, only configure tracing events. Meant to be used with customized event log writer.

-L ⟨num⟩
Sets the maximum length of a cost-centre stack name in a heap profile. Defaults to 25.

8.5.2 Retainer Profiling

Retainer profiling is designed to help answer questions like "why is this data being retained?". We start by defining what we mean by a retainer:

> A retainer is either the system stack, an unevaluated closure (thunk), or an explicitly mutable object.

In particular, constructors are *not* retainers.

An object `B` retains object `A` if (i) `B` is a retainer object and (ii) object `A` can be reached by recursively following pointers starting from object `B`, but not meeting any other retainer objects on the way. Each live object is retained by one or more retainer objects, collectively called its retainer set, or its retainer set, or its retainers.

When retainer profiling is requested by giving the program the `-hr` option, a graph is generated which is broken down by retainer set. A retainer set is displayed as a set of cost-centre stacks; because this is usually too large to fit on the profile graph, each retainer set is numbered and shown abbreviated on the graph along with its number, and the full list of retainer sets is dumped into the file `prog.prof`.

Retainer profiling requires multiple passes over the live heap in order to discover the full retainer set for each object, which can be quite slow. So we set a limit on the maximum size of a retainer set, where all retainer sets larger than the maximum retainer set size are replaced by the special set `MANY`. The maximum set size defaults to 8 and can be altered with the *-R ⟨size⟩* (page 668) RTS option:

-R ⟨size⟩
: Restrict the number of elements in a retainer set to ⟨size⟩ (default 8).

8.5.2.1 Hints for using retainer profiling

The definition of retainers is designed to reflect a common cause of space leaks: a large structure is retained by an unevaluated computation, and will be released once the computation is forced. A good example is looking up a value in a finite map, where unless the lookup is forced in a timely manner the unevaluated lookup will cause the whole mapping to be retained. These kind of space leaks can often be eliminated by forcing the relevant computations to be performed eagerly, using `seq` or strictness annotations on data constructor fields.

Often a particular data structure is being retained by a chain of unevaluated closures, only the nearest of which will be reported by retainer profiling - for example `A` retains `B`, `B` retains `C`, and `C` retains a large structure. There might be a large number of `B`s but only a single `A`, so `A` is really the one we're interested in eliminating. However, retainer profiling will in this case report `B` as the retainer of the large structure. To move further up the chain of retainers, we can ask for another retainer profile but this time restrict the profile to `B` objects, so we get a profile of the retainers of `B`:

```
prog +RTS -hr -hcB
```

This trick isn't foolproof, because there might be other `B` closures in the heap which aren't the retainers we are interested in, but we've found this to be a useful technique in most cases.

8.5.3 Precise Retainer Analysis

If you want to precisely answer questions about why a certain type of closure is retained then it is worthwhile using ghc-debug which has a terminal interface which can be used to easily answer queries such as, what is retaining a certain closure.

8.5.4 Biographical Profiling

A typical heap object may be in one of the following four states at each point in its lifetime:

- The lag stage, which is the time between creation and the first use of the object,
- the use stage, which lasts from the first use until the last use of the object, and
- The drag stage, which lasts from the final use until the last reference to the object is dropped.
- An object which is never used is said to be in the void state for its whole lifetime.

A biographical heap profile displays the portion of the live heap in each of the four states listed above. Usually the most interesting states are the void and drag states: live heap in these states is more likely to be wasted space than heap in the lag or use states.

It is also possible to break down the heap in one or more of these states by a different criteria, by restricting a profile by biography. For example, to show the portion of the heap in the drag or void state by producer:

```
prog +RTS -hc -hbdrag,void
```

Once you know the producer or the type of the heap in the drag or void states, the next step is usually to find the retainer(s):

```
prog +RTS -hr -hccc...
```

Note: This two stage process is required because GHC cannot currently profile using both biographical and retainer information simultaneously.

8.5.5 Actual memory residency

How does the heap residency reported by the heap profiler relate to the actual memory residency of your program when you run it? You might see a large discrepancy between the residency reported by the heap profiler, and the residency reported by tools on your system (eg. `ps` or `top` on Unix, or the Task Manager on Windows). There are several reasons for this:

- There is an overhead of profiling itself, which is subtracted from the residency figures by the profiler. This overhead goes away when compiling without profiling support, of course. The space overhead is currently 2 extra words per heap object, which probably results in about a 30% overhead.
- Garbage collection requires more memory than the actual residency. The factor depends on the kind of garbage collection algorithm in use: a major GC in the standard generation copying collector will usually require $3L$ bytes of memory, where L is the amount of live data. This is because by default (see the RTS `-F ⟨factor⟩` (page 186) option) we allow the old generation to grow to twice its size ($2L$) before collecting it, and we require additionally L bytes to copy the live data into. When using compacting collection (see

the *-c* (page 186) option), this is reduced to $2L$, and can further be reduced by tweaking the *-F ⟨factor⟩* (page 186) option. Also add the size of the allocation area (see *-A ⟨size⟩* (page 185)).

- The program text itself, the C stack, any non-heap data (e.g. data allocated by foreign libraries, and data allocated by the RTS), and `mmap()`'d memory are not counted in the heap profile.

For more discussion about understanding how understanding process residency see *Understanding how OS memory usage corresponds to live data* (page 696).

8.6 hp2ps – Rendering heap profiles to PostScript

Usage:

```
hp2ps [flags] [<file>[.hp]]
```

The program **hp2ps** program converts a `.hp` file produced by the `-h<break-down>` runtime option into a PostScript graph of the heap profile. By convention, the file to be processed by **hp2ps** has a `.hp` extension. The PostScript output is written to `file@.ps`. If `<file>` is omitted entirely, then the program behaves as a filter.

hp2ps is distributed in `ghc/utils/hp2ps` in a GHC source distribution. It was originally developed by Dave Wakeling as part of the HBC/LML heap profiler.

The flags are:

-d
: In order to make graphs more readable, `hp2ps` sorts the shaded bands for each identifier. The default sort ordering is for the bands with the largest area to be stacked on top of the smaller ones. The `-d` option causes rougher bands (those representing series of values with the largest standard deviations) to be stacked on top of smoother ones.

-b
: Normally, `hp2ps` puts the title of the graph in a small box at the top of the page. However, if the JOB string is too long to fit in a small box (more than 35 characters), then `hp2ps` will choose to use a big box instead. The `-b` option forces `hp2ps` to use a big box.

-e⟨float⟩[in|mm|pt]
: Generate encapsulated PostScript suitable for inclusion in LaTeX documents. Usually, the PostScript graph is drawn in landscape mode in an area 9 inches wide by 6 inches high, and `hp2ps` arranges for this area to be approximately centred on a sheet of a4 paper. This format is convenient of studying the graph in detail, but it is unsuitable for inclusion in LaTeX documents. The `-e` option causes the graph to be drawn in portrait mode, with float specifying the width in inches, millimetres or points (the default). The resulting PostScript file conforms to the Encapsulated PostScript (EPS) convention, and it can be included in a LaTeX document using Rokicki's dvi-to-PostScript converter `dvips`.

-g
: Create output suitable for the `gs` PostScript previewer (or similar). In this case the graph is printed in portrait mode without scaling. The output is unsuitable for a laser printer.

-l
: Normally a profile is limited to 20 bands with additional identifiers being grouped into an `OTHER` band. The `-l` flag removes this 20 band and limit, producing as many bands as necessary. No key is produced as it won't fit!. It is useful for creation time profiles with many bands.

-m⟨int⟩
Normally a profile is limited to 20 bands with additional identifiers being grouped into an OTHER band. The -m flag specifies an alternative band limit (the maximum is 20).

-m0 requests the band limit to be removed. As many bands as necessary are produced. However no key is produced as it won't fit! It is useful for displaying creation time profiles with many bands.

-p
Use previous parameters. By default, the PostScript graph is automatically scaled both horizontally and vertically so that it fills the page. However, when preparing a series of graphs for use in a presentation, it is often useful to draw a new graph using the same scale, shading and ordering as a previous one. The -p flag causes the graph to be drawn using the parameters determined by a previous run of hp2ps on file. These are extracted from file@.aux.

-s
Use a small box for the title.

-t⟨float⟩
Normally trace elements which sum to a total of less than 1% of the profile are removed from the profile. The -t option allows this percentage to be modified (maximum 5%).

-t0 requests no trace elements to be removed from the profile, ensuring that all the data will be displayed.

-c
Generate colour output.

-y
Ignore marks.

-?
Print out usage information.

8.7 Profiling Parallel and Concurrent Programs

Combining *-threaded* (page 248) and *-prof* (page 656) is perfectly fine, and indeed it is possible to profile a program running on multiple processors with the RTS *-N ⟨x⟩* (page 132) option.[3]

Some caveats apply, however. In the current implementation, a profiled program is likely to scale much less well than the unprofiled program, because the profiling implementation uses some shared data structures which require locking in the runtime system. Furthermore, the memory allocation statistics collected by the profiled program are stored in shared memory but *not* locked (for speed), which means that these figures might be inaccurate for parallel programs.

We strongly recommend that you use *-fno-prof-count-entries* (page 656) when compiling a program to be profiled on multiple cores, because the entry counts are also stored in shared memory, and continuously updating them on multiple cores is extremely slow.

We also recommend using ThreadScope for profiling parallel programs; it offers a GUI for visualising parallel execution, and is complementary to the time and space profiling features provided with GHC.

[3] This feature was added in GHC 7.4.1.

8.8 Observing Code Coverage

Code coverage tools allow a programmer to determine what parts of their code have been actually executed, and which parts have never actually been invoked. GHC has an option for generating instrumented code that records code coverage as part of the Haskell Program Coverage (HPC) toolkit, which is included with GHC. HPC tools can be used to render the generated code coverage information into human understandable format.

Correctly instrumented code provides coverage information of two kinds: source coverage and boolean-control coverage. Source coverage is the extent to which every part of the program was used, measured at three different levels: declarations (both top-level and local), alternatives (among several equations or case branches) and expressions (at every level). Boolean coverage is the extent to which each of the values True and False is obtained in every syntactic boolean context (ie. guard, condition, qualifier).

HPC displays both kinds of information in two primary ways: textual reports with summary statistics (`hpc report`) and sources with color mark-up (`hpc markup`). For boolean coverage, there are four possible outcomes for each guard, condition or qualifier: both True and False values occur; only True; only False; never evaluated. In hpc-markup output, highlighting with a yellow background indicates a part of the program that was never evaluated; a green background indicates an always-True expression and a red background indicates an always-False one.

8.8.1 A small example: Reciprocation

For an example we have a program, called `Recip.hs`, which computes exact decimal representations of reciprocals, with recurring parts indicated in brackets.

```haskell
reciprocal :: Int -> (String, Int)
reciprocal n | n > 1 = ('0' : '.' : digits, recur)
             | otherwise = error
              "attempting to compute reciprocal of number <= 1"
  where
  (digits, recur) = divide n 1 []
divide :: Int -> Int -> [Int] -> (String, Int)
divide n c cs | c `elem` cs = ([], position c cs)
              | r == 0      = (show q, 0)
              | r /= 0      = (show q ++ digits, recur)
  where
  (q, r) = (c*10) `quotRem` n
  (digits, recur) = divide n r (c:cs)

position :: Int -> [Int] -> Int
position n (x:xs) | n==x      = 1
                  | otherwise = 1 + position n xs

showRecip :: Int -> String
showRecip n =
  "1/" ++ show n ++ " = " ++
  if r==0 then d else take p d ++ "(" ++ drop p d ++ ")"
  where
  p = length d - r
  (d, r) = reciprocal n

main = do
```

(continues on next page)

(continued from previous page)

```
  number <- readLn
  putStrLn (showRecip number)
  main
```

HPC instrumentation is enabled with the *-fhpc* (page 674) flag:

```
$ ghc -fhpc Recip.hs
```

GHC creates a subdirectory `.hpc` in the current directory, and puts HPC index (`.mix`) files in there, one for each module compiled. You don't need to worry about these files: they contain information needed by the `hpc` tool to generate the coverage data for compiled modules after the program is run.

```
$ ./Recip
1/3
= 0.(3)
```

Running the program generates a file with the `.tix` suffix, in this case `Recip.tix`, which contains the coverage data for this run of the program. The program may be run multiple times (e.g. with different test data), and the coverage data from the separate runs is accumulated in the `.tix` file. To reset the coverage data and start again, just remove the `.tix` file. You can control where the `.tix` file is generated using the environment variable *HPCTIXFILE* (page 673).

HPCTIXFILE
: Set the HPC `.tix` file output path.

Having run the program, we can generate a textual summary of coverage:

```
$ hpc report Recip
 80% expressions used (81/101)
 12% boolean coverage (1/8)
      14% guards (1/7), 3 always True,
                        1 always False,
                        2 unevaluated
       0% 'if' conditions (0/1), 1 always False
     100% qualifiers (0/0)
 55% alternatives used (5/9)
100% local declarations used (9/9)
100% top-level declarations used (5/5)
```

We can also generate a marked-up version of the source.

```
$ hpc markup Recip
writing Recip.hs.html
```

This generates one file per Haskell module, and 4 index files, `hpc_index.html`, `hpc_index_alt.html`, `hpc_index_exp.html`, `hpc_index_fun.html`.

8.8.2 Options for instrumenting code for coverage

-fhpc

Enable code coverage for the current module or modules being compiled.

Modules compiled with this option can be freely mixed with modules compiled without it; indeed, most libraries will typically be compiled without *-fhpc* (page 674). When the program is run, coverage data will only be generated for those modules that were compiled with *-fhpc* (page 674), and the **hpc** tool will only show information about those modules.

-hpcdir⟨dir⟩

Default .hpc

Override the directory where GHC places the HPC index (`.mix`) files used by `hpc` to understand program structure.

8.8.3 The hpc toolkit

The hpc command has several sub-commands:

```
$ hpc
Usage: hpc COMMAND ...

Commands:
  help        Display help for hpc or a single command
Reporting Coverage:
  report      Output textual report about program coverage
  markup      Markup Haskell source with program coverage
Processing Coverage files:
  sum         Sum multiple .tix files in a single .tix file
  combine     Combine two .tix files in a single .tix file
  map         Map a function over a single .tix file
Coverage Overlays:
  overlay     Generate a .tix file from an overlay file
  draft       Generate draft overlay that provides 100% coverage
Others:
  show        Show .tix file in readable, verbose format
  version     Display version for hpc
```

In general, these options act on a `.tix` file after an instrumented binary has generated it.

The hpc tool assumes you are in the top-level directory of the location where you built your application, and the `.tix` file is in the same top-level directory. You can use the flag `--srcdir` to use `hpc` for any other directory, and use `--srcdir` multiple times to analyse programs compiled from difference locations, as is typical for packages.

We now explain in more details the major modes of hpc.

8.8.3.1 hpc report

`hpc report` gives a textual report of coverage. By default, all modules and packages are considered in generating report, unless include or exclude are used. The report is a summary unless the `--per-module` flag is used. The `--xml-output` option allows for tools to use hpc to glean coverage.

```
$ hpc help report
Usage: hpc report [OPTION] .. <TIX_FILE> [<MODULE> [<MODULE> ..]]

Options:

    --per-module                  show module level detail
    --decl-list                   show unused decls
    --exclude=[PACKAGE:][MODULE]  exclude MODULE and/or PACKAGE
    --include=[PACKAGE:][MODULE]  include MODULE and/or PACKAGE
    --srcdir=DIR                  path to source directory of .hs files
                                  multi-use of srcdir possible
    --hpcdir=DIR                  append sub-directory that contains .mix files
                                  default .hpc [rarely used]
    --reset-hpcdirs               empty the list of hpcdir's
                                  [rarely used]
    --xml-output                  show output in XML
```

8.8.3.2 hpc markup

`hpc markup` marks up source files into colored html.

```
$ hpc help markup
Usage: hpc markup [OPTION] .. <TIX_FILE> [<MODULE> [<MODULE> ..]]

Options:

    --exclude=[PACKAGE:][MODULE]  exclude MODULE and/or PACKAGE
    --include=[PACKAGE:][MODULE]  include MODULE and/or PACKAGE
    --srcdir=DIR                  path to source directory of .hs files
                                  multi-use of srcdir possible
    --hpcdir=DIR                  append sub-directory that contains .mix files
                                  default .hpc [rarely used]
    --reset-hpcdirs               empty the list of hpcdir's
                                  [rarely used]
    --fun-entry-count             show top-level function entry counts
    --highlight-covered           highlight covered code, rather that code gaps
    --destdir=DIR                 path to write output to
```

8.8.3.3 hpc sum

`hpc sum` adds together any number of `.tix` files into a single `.tix` file. `hpc sum` does not change the original `.tix` file; it generates a new `.tix` file.

```
$ hpc help sum
Usage: hpc sum [OPTION] .. <TIX_FILE> [<TIX_FILE> [<TIX_FILE> ..]]
Sum multiple .tix files in a single .tix file

Options:

    --exclude=[PACKAGE:][MODULE]  exclude MODULE and/or PACKAGE
    --include=[PACKAGE:][MODULE]  include MODULE and/or PACKAGE
    --output=FILE                 output FILE
    --union                       use the union of the module namespace (default is␣
↪intersection)
```

8.8.3.4 hpc combine

`hpc combine` is the swiss army knife of `hpc`. It can be used to take the difference between `.tix` files, to subtract one `.tix` file from another, or to add two `.tix` files. hpc combine does not change the original `.tix` file; it generates a new `.tix` file.

```
$ hpc help combine
Usage: hpc combine [OPTION] .. <TIX_FILE> <TIX_FILE>
Combine two .tix files in a single .tix file

Options:

    --exclude=[PACKAGE:][MODULE]  exclude MODULE and/or PACKAGE
    --include=[PACKAGE:][MODULE]  include MODULE and/or PACKAGE
    --output=FILE                 output FILE
    --function=FUNCTION           combine .tix files with join function, default = ADD
                                  FUNCTION = ADD | DIFF | SUB
    --union                       use the union of the module namespace (default is␣
↪intersection)
```

8.8.3.5 hpc map

hpc map inverts or zeros a `.tix` file. hpc map does not change the original `.tix` file; it generates a new `.tix` file.

```
$ hpc help map
Usage: hpc map [OPTION] .. <TIX_FILE>
Map a function over a single .tix file

Options:

    --exclude=[PACKAGE:][MODULE]  exclude MODULE and/or PACKAGE
    --include=[PACKAGE:][MODULE]  include MODULE and/or PACKAGE
    --output=FILE                 output FILE
    --function=FUNCTION           apply function to .tix files, default = ID
                                  FUNCTION = ID | INV | ZERO
    --union                       use the union of the module namespace (default is␣
↪intersection)
```

8.8.3.6 hpc overlay and hpc draft

Overlays are an experimental feature of HPC, a textual description of coverage. hpc draft is used to generate a draft overlay from a .tix file, and hpc overlay generates a .tix files from an overlay.

```
% hpc help overlay
Usage: hpc overlay [OPTION] .. <OVERLAY_FILE> [<OVERLAY_FILE> [...]]

Options:

    --srcdir=DIR   path to source directory of .hs files
                   multi-use of srcdir possible
    --hpcdir=DIR                  append sub-directory that contains .mix files
                                  default .hpc [rarely used]
    --reset-hpcdirs               empty the list of hpcdir's
                                  [rarely used]
    --output=FILE  output FILE
% hpc help draft
Usage: hpc draft [OPTION] .. <TIX_FILE>

Options:

    --exclude=[PACKAGE:][MODULE]  exclude MODULE and/or PACKAGE
    --include=[PACKAGE:][MODULE]  include MODULE and/or PACKAGE
    --srcdir=DIR                  path to source directory of .hs files
                                  multi-use of srcdir possible
    --hpcdir=DIR                  append sub-directory that contains .mix files
                                  default .hpc [rarely used]
    --reset-hpcdirs               empty the list of hpcdir's
                                  [rarely used]
    --output=FILE                 output FILE
```

8.8.4 Caveats and Shortcomings of Haskell Program Coverage

HPC does not attempt to lock the `.tix` file, so multiple concurrently running binaries in the same directory will exhibit a race condition. At compile time, there is no way to change the name of the `.tix` file generated; at runtime, the name of the generated `.tix` file can be changed using *HPCTIXFILE* (page 673); the name of the `.tix` file will also change if you rename the binary. HPC does not work with GHCi.

8.9 Using "ticky-ticky" profiling (for implementors)

-ticky

Enable ticky-ticky profiling. By default this only tracks the allocations *by* each closure type. See *-ticky-allocd* (page 678) to keep track of allocations *of* each closure type as well.

GHC's ticky-ticky profiler provides a low-level facility for tracking entry and allocation counts of particular individual closures. Ticky-ticky profiling requires a certain familiarity with GHC internals, so it is best suited for expert users, but can provide an invaluable precise insight into the allocation behaviour of your programs.

Getting started with ticky profiling consists of three steps.

1. Add the `-ticky` flag when compiling a Haskell module to enable "ticky-ticky" profiling of that module. This makes GHC emit performance-counting instructions in every STG function.
2. Add `-ticky` to the command line when linking, so that you link against a version of the runtime system that allows you to display the results. In fact, in the link phase -ticky implies -debug, so you get the debug version of the runtime system too.
3. Then when running your program you can collect the results of the profiling in two ways.

- Using the eventlog, the *-lT* (page 195) flag will emit ticky samples to the eventlog periodically. This has the advantage of being able to resolve dynamic behaviors over the program's lifetime. See *Ticky counters* (page 759) for details on the event types reported. The ticky information can be rendered into an interactive table using eventlog2html.
- A legacy textual format is emitted using the *-r ⟨file⟩* (page 198) flag. This produces a textual table containing information about how much each counter ticked throughout the duration of the program.

8.9.1 Additional Ticky Flags

There are some additional flags which can be used to increase the number of ticky counters and the quality of the profile.

-ticky-allocd
Keep track of how much each closure type is allocated.

-ticky-dyn-thunk
Track allocations of dynamic thunks.

-ticky-LNE
These are not allocated, and can be very performance sensitive so we usually don't want to run ticky counters for these to avoid even worse performance for tickied builds.

But sometimes having information about these binders is critical. So we have a flag to ticky them anyway.

-ticky-tag-checks
These dummy counters contain:

- The number of avoided tag checks in the entry count.
- "infer" as the argument string to distinguish them from regular counters.
- The name of the variable we are casing on, as well as a unique to represent the inspection site as one variable might be cased on multiple times. The unique comes first with the variable coming at the end. Like this: `u10_s98c (Main) at nofib/spectral/simple/Main.hs:677:1 in u10` where *u10* is the variable and *u10_s98c* the unique associated with the inspection site.

Note that these counters are currently not processed well be eventlog2html. So if you want to check them you will have to use the text based interface.

-ticky-ap-thunk
This allows us to get accurate entry counters for code like *f x y* at the cost of code size. We do this but not using the precomputed standard AP thunk code.

GHC's ticky-ticky profiler provides a low-level facility for tracking entry and allocation counts of particular individual closures. Because ticky-ticky profiling requires a certain familiarity with GHC internals, we have moved the documentation to the GHC developers wiki. Take a look at its overview of the profiling options, which includes a link to the ticky-ticky profiling page.

Note that ticky-ticky samples can be emitted in two formats: the eventlog, using the *-lT* (page 195) event type, and a plain text summary format, using the *-r ⟨file⟩* (page 198) option. The former has the advantage of being able to resolve dynamic behaviors over the program's lifetime. See *Ticky counters* (page 759) for details on the event types reported.

8.9.2 Understanding the Output of Ticky-Ticky profiles

Once you have your rendered profile then you can begin to understand the allocation behaviour of your program. There are two classes of ticky-ticky counters.

Name-specific counters

> Each "name-specific counter" is associated with a name that is defined in the result of the optimiser. For each such name, there are three possible counters: entries, heap allocation by the named thing, and heap used to allocate that named thing.

Global counters

> Each "global counter" describes some aspect of the entire program execution. For example, one global counter tracks total heap allocation; another tracks allocation for PAPs.

In general you are probably interested mostly in the name-specific counters as these can provided detailed information about where allocates how much in your program.

8.9.3 Information about name-specific counters

Name-specific counters provide the following information about a closure.

- Entries - How many times the closure was entered.
- Allocs - How much (in bytes) is allocated *by* that closure.
- Allod - How often the closure is allocated.
- FVs - The free variables captured by that closure.
- Args - The arguments that closure takes.

The FVs and Args information is encoded using a small DSL.

Classification	Description
`+`	dictionary
`\>`	function
`{C,I,F,D,W}`	char, int, float, double, word
`{c,i,f,d,w}`	unboxed ditto
`T`	unboxed tuple
`P`	other primitive type
`p`	unboxed primitive type
`L`	list
`E`	enumeration type
`S`	single-constructor type
`M`	multi-constructor type
`.`	other type
`-`	reserved for others to mark as "uninteresting"

In particular note that you can use the ticky profiler to see any function calls to dictionary arguments by searching the profile for the + classifier. This indicates that the function has failed to specialise for one reason or another.

8.9.4 Examples

A typical use of ticky-ticky would be to generate a ticky report using the eventlog by evoking an application with RTS arguments like this:

```
app <args> +RTS -l-augT
```

This will produce an eventlog file which contains results from ticky counters. This file can be manually inspected like any regular eventlog. However for ticky-ticky eventlog2html has good support for producing tables from these logs.

With an up to date version of eventlog2html this can be simply done by invoking eventlog2html on the produced eventlog. In the example above the invocation would then be `eventlog2html app.eventlog` Which will produce a searchable and sortable table containing all the ticky counters in the log.

8.9.5 Notes about ticky profiling

- You can mix together modules compiled with and without `-ticky` but you will miss out on allocations and counts from uninstrumented modules in the profile.
- Linking with the `-ticky` has a quite severe performance impact on your program. `-ticky` implies using the unoptimised `-debug` RTS. Therefore `-ticky` shouldn't be used for production builds.
- Building with `-ticky` doesn't affect core optimisations of your program as the counters are inserted after the STG pipeline. At which point most optimizations have already been run.
- When using the eventlog it is possible to combine together ticky-ticky and IPE based profiling as each ticky counter definition has an associated info table. This address can be looked up in the IPE map so that further information (such as source location) can be determined about that closure.
- Global ticky counters are only available in the textual ticky output (`+RTS -r`). But this mode has some limitations (e.g. on column widths) and will contain raw json output in some columns. For this reason using an eventlog-based approach should be prefered if possible.

CHAPTER

NINE

DEBUGGING COMPILED PROGRAMS

Since the 7.10 release GHC can emit debugging information to help debugging tools understand the code that GHC produces. This debugging information is usable by most UNIX debugging tools.

-g
-g⟨n⟩

> **Since** 7.10, numeric levels since 8.0
>
> **Implies** *-fexpose-internal-symbols* (page 245) when ⟨n⟩ >= 2.

Emit debug information in object code. Currently only DWARF debug information is supported on x86-64 and i386. Currently debug levels 0 through 3 are accepted:

- `-g0`: no debug information produced
- `-g1`: produces stack unwinding records for top-level functions (sufficient for basic backtraces)
- `-g2`: produces stack unwinding records for top-level functions as well as inner blocks (allowing more precise backtraces than with `-g1`).
- `-g3`: produces GHC-specific DWARF information for use by more sophisticated Haskell-aware debugging tools (see *Debugging information entities* (page 686) for details)

If ⟨n⟩ is omitted, level 2 is assumed.

Note that for stack unwinding to be reliable, all libraries, including foreign libraries and those shipped with GHC such as `base`, must be compiled with unwinding information. GHC binary distributions configured in this way are provided for a select number of platforms; other platforms are advised to build using Hadrian's `+debug_info` flavour transformer. Note as well that the built-in unwinding support provided by the `base` library's GHC.ExecutionStack module requires that the runtime system be built with `libdw` support enabled (using the `--enable-dwarf-unwind` flag to `configure` while building the compiler) and a platform which `libdw` supports.

9.1 Tutorial

Let's consider a simple example,

```haskell
-- fib.hs
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

main :: IO ()
main = print $ fib 50
```

Let's first see how execution flows through this program. We start by telling GHC that we want debug information,

```
$ ghc -g -rtsopts fib.hs
```

Here we used the `-g` option to inform GHC that it should add debugging information in the produced binary. There are three levels of debugging output: `-g0` (no debugging information, the default), `-g1` (sufficient for basic backtraces), `-g2` (or just `-g` for short; emitting everything GHC knows). Note that this debugging information does not affect the optimizations performed by GHC.

Tip: Under Mac OS X debug information is kept apart from the executable. After compiling the executable you'll need to use the `dsymutil` utility to extract the debugging information and place them in the debug archive,

```
$ dsymutil fib
```

This should produce a file named `fib.dSYM`.

Now let's have a look at the flow of control. For this we can just start our program under `gdb` (or an equivalent debugger) as we would any other native executable,

```
$ gdb --args ./Fib +RTS -V0
Reading symbols from Fib...done.
(gdb) run
Starting program: /opt/exp/ghc/ghc-dwarf/Fib
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
^C
Program received signal SIGINT, Interrupt.
0x000000000064fc7c in cfy4_info () at libraries/integer-gmp/src/GHC/Integer/Type.
↪hs:424
424     minusInteger x y = inline plusInteger x (inline negateInteger y)
(gdb)
```

Here we have used the runtime system's `-V0` option to disable the RTS's periodic timer which may interfere with our debugging session. Upon breaking into the program `gdb` shows us a location in our source program corresponding to the current point of execution.

Moreover, we can ask `gdb` to tell us the flow of execution that lead us to this point in the program,

```
(gdb) bt
#0  0x000000000064fc7c in cfy4_info () at libraries/integer-gmp/src/GHC/Integer/Type.hs:424
#1  0x00000000006eb0c0 in ?? ()
#2  0x000000000064301c in cbuV_info () at libraries/integer-gmp/src/GHC/Integer/Type.hs:323
#3  0x000000000064311b in integerzmgmp_GHCziIntegerziType_eqInteger_info () at libraries/integer-gmp/src/GHC/Integer/Type.hs:312
#4  0x0000000000406eca in roz_info () at Fib.hs:2
#5  0x00000000006eb0c0 in ?? ()
#6  0x000000000064f075 in cfru_info () at libraries/integer-gmp/src/GHC/Integer/Type.hs:412
#7  0x00000000006eb0c0 in ?? ()
#8  0x000000000064f075 in cfru_info () at libraries/integer-gmp/src/GHC/Integer/Type.hs:412
#9  0x00000000006eb0c0 in ?? ()
#10 0x000000000064eefe in integerzmgmp_GHCziIntegerziType_plusInteger_info () at libraries/integer-gmp/src/GHC/Integer/Type.hs:393
...
#64 0x0000000000643ac8 in integerzmgmp_GHCziIntegerziType_ltIntegerzh_info () at libraries/integer-gmp/src/GHC/Integer/Type.hs:343
#65 0x00000000004effcc in base_GHCziShow_zdwintegerToString_info () at libraries/base/GHC/Show.hs:443
#66 0x00000000004f0795 in base_GHCziShow_zdfShowIntegerzuzdcshow_info () at libraries/base/GHC/Show.hs:145
#67 0x000000000048892b in cdGW_info () at libraries/base/GHC/IO/Handle/Text.hs:595
#68 0x0000000000419cb2 in base_GHCziBase_thenIO1_info () at libraries/base/GHC/Base.hs:1072
```

Hint: Here we notice the first bit of the stack trace has many unidentified stack frames at address `0x006eb0c0`. If we ask `gdb` about this location, we find that these frames are actually STG update closures,

```
(gdb) print/a 0x006eb0c0
$1 = 0x6eb0c0 <stg_upd_frame_info>
```

The reason `gdb` doesn't show this symbol name in the backtrace output is an infidelity in its interpretation of debug information, which assumes an invariant preserved in C but not Haskell programs. Unfortunately it is necessary to work around this manually until this behavior is fixed upstream.

Note: Because of the aggressive optimization that GHC performs to the programs it compiles it is quite difficult to pin-point exactly which point in the source program a given machine instruction should be attributed to. In fact, internally GHC associates each instruction with a **set** of source locations. When emitting the standard debug information used by `gdb` and other language-agnostic debugging tools, GHC is forced to heuristically choose one location from among this set.

For this reason we should be cautious when interpreting the source locations provided by GDB. While these locations will usually be in some sense "correct", they aren't always useful. This is why profiling tools targeting Haskell should supplement the standard source location information with GHC-specific annotations (emitted with `-g2`) when assigning costs.

Indeed, we can even set breakpoints,

```
(gdb) break fib.hs:4
Breakpoint 1 at 0x406c60: fib.hs:4. (5 locations)
(gdb) run
Starting program: /opt/exp/ghc/ghc-dwarf/Fib

Breakpoint 1, c1RV_info () at Fib.hs:4
4        fib n = fib (n-1) + fib (n-2)
(gdb) bt
#0  c1RV_info () at Fib.hs:4
#1  0x00000000006eb0c0 in ?? ()
#2  0x0000000000643ac8 in integerzmgmp_GHCziIntegerziType_ltIntegerzh_info () at␣
↪libraries/integer-gmp/src/GHC/Integer/Type.hs:343
#3  0x00000000004effcc in base_GHCziShow_zdwintegerToString_info () at libraries/base/
↪GHC/Show.hs:443
#4  0x00000000004f0795 in base_GHCziShow_zdfShowIntegerzuzdcshow_info () at libraries/
↪base/GHC/Show.hs:145
#5  0x000000000048892b in cdGW_info () at libraries/base/GHC/IO/Handle/Text.hs:595
#6  0x0000000000419cb2 in base_GHCziBase_thenIO1_info () at libraries/base/GHC/Base.
↪hs:1072
#7  0x00000000006ebcb0 in ?? () at rts/Exception.cmm:332
#8  0x00000000006e7320 in ?? ()
(gdb)
```

Due to the nature of GHC's heap and the heavy optimization that it performs, it is quite difficult to probe the values of bindings at runtime. In this way, the debugging experience of a Haskell program with DWARF support is still a bit impoverished compared to typical imperative debuggers.

9.2 Requesting a stack trace from Haskell code

GHC's runtime system has built-in support for collecting stack trace information from a running Haskell program. This currently requires that the `libdw` library from the `elfutils` package is available. Of course, the backtrace will be of little use unless debug information is available in the executable and its dependent libraries.

Stack trace functionality is exposed for use by Haskell programs in the GHC.ExecutionStack module. See the Haddock documentation in this module for details regarding usage.

9.3 Requesting a stack trace with `SIGQUIT`

On POSIX-compatible platforms GHC's runtime system (when built with `libdw` support) will produce a stack trace on `stderr` when a `SIGQUIT` signal is received (on many systems this signal can be sent using `Ctrl-\`). For instance (using the same `fib.hs` as above),

```
$ ./fib  &  killall -SIGQUIT fib

Caught SIGQUIT; Backtrace:
0x7f3176b15dd8    dwfl_thread_getframes (/usr/lib/x86_64-linux-gnu/libdw-0.163.so)
0x7f3176b1582f    (null) (/usr/lib/x86_64-linux-gnu/libdw-0.163.so)
0x7f3176b15b57    dwfl_getthreads (/usr/lib/x86_64-linux-gnu/libdw-0.163.so)
0x7f3176b16150    dwfl_getthread_frames (/usr/lib/x86_64-linux-gnu/libdw-0.163.so)
      0x6dc857    libdwGetBacktrace (rts/Libdw.c:248.0)
```

(continues on next page)

(continued from previous page)

```
      0x6e6126    backtrace_handler (rts/posix/Signals.c:541.0)
0x7f317677017f    (null) (/lib/x86_64-linux-gnu/libc-2.19.so)
      0x642e1c    integerzmgmp_GHCziIntegerziType_eqIntegerzh_info (libraries/integer-
↪gmp/src/GHC/Integer/Type.hs:320.1)
      0x643023    integerzmgmp_GHCziIntegerziType_eqInteger_info (libraries/integer-
↪gmp/src/GHC/Integer/Type.hs:312.1)
      0x406eca    roz_info (/opt/exp/ghc/ghc-dwarf//Fib.hs:2.1)
      0x6eafc0    stg_upd_frame_info (rts/Updates.cmm:31.1)
      0x64ee06    integerzmgmp_GHCziIntegerziType_plusInteger_info (libraries/integer-
↪gmp/src/GHC/Integer/Type.hs:393.1)
      0x6eafc0    stg_upd_frame_info (rts/Updates.cmm:31.1)
...
      0x6439d0    integerzmgmp_GHCziIntegerziType_ltIntegerzh_info (libraries/integer-
↪gmp/src/GHC/Integer/Type.hs:343.1)
      0x4efed4    base_GHCziShow_zdwintegerToString_info (libraries/base/GHC/Show.
↪hs:442.1)
      0x4f069d    base_GHCziShow_zdfShowIntegerzuzdcshow_info (libraries/base/GHC/Show.
↪hs:145.5)
      0x488833    base_GHCziIOziHandleziText_zdwa8_info (libraries/base/GHC/IO/Handle/
↪Text.hs:582.1)
      0x6ebbb0    stg_catch_frame_info (rts/Exception.cmm:370.1)
      0x6e7220    stg_stop_thread_info (rts/StgStartup.cmm:42.1)
```

9.4 Implementor's notes: DWARF annotations

Note: Most users don't need to worry about the details described in this section. This discussion is primarily targeted at tooling authors who need to interpret the GHC-specific DWARF annotations contained in compiled binaries.

When invoked with the `-g` flag GHC will produce standard DWARF v4 debugging information. This format is used by nearly all POSIX-compliant targets and can be used by debugging and performance tools (e.g. `gdb`, `lldb`, and `perf`) to understand the structure of GHC-compiled programs.

In particular GHC produces the following DWARF sections,

`.debug_info` Debug information entities (DIEs) describing all of the basic blocks in the compiled program.

`.debug_line` Line number information necessary to map instruction addresses to line numbers in the source program.

Note that the line information in this section is not nearly as rich as the information provided in `.debug_info`. Whereas `.debug_line` requires that each instruction is assigned exactly one source location, the DIEs in `.debug_info` can be used to identify all relevant sources locations.

`.debug_frames` Call frame information (CFI) necessary for stack unwinding to produce a call stack trace.

`.debug_arange` Address range information necessary for efficient lookup in debug information.

9.4.1 Debugging information entities

GHC may produce the following standard DIEs in the `.debug_info` section,

`DW_TAG_compile_unit` Represents a compilation unit (e.g. a Haskell module).

`DW_TAG_subprogram` Represents a C-- top-level basic block.

`DW_TAG_lexical_block` Represents a C-- basic block. Note that this is a slight departure from the intended meaning of this DIE type as it does not necessarily reflect lexical scope in the source program.

As GHC's compilation products don't map perfectly onto DWARF constructs, GHC takes advantage of the extensibility of the DWARF standard to provide additional information.

Unfortunately DWARF isn't expressive enough to fully describe the code that GHC produces. This is most apparent in the case of line information, where GHC is forced to choose some between a variety of possible originating source locations. This limits the usefulness of DWARF information with traditional statistical profiling tools. For profiling it is recommended that one use the extended debugging information. See the *Profiling* section below.

In addition to the usual DIEs specified by the DWARF specification, GHC produces a variety of others using the vendor-extensibility regions of the tag and attribute space.

9.4.1.1 `DW_TAG_ghc_src_note`

`DW_TAG_ghc_src_note` DIEs (tag 0x5b01) are found as children of `DW_TAG_lexical_block` DIEs. They describe source spans which gave rise to the block; formally these spans are causally responsible for produced code: changes to code in the given span may change the code within the block; conversely changes outside the span are guaranteed not to affect the code in the block.

Spans are described with the following attributes,

`DW_AT_ghc_span_file` (0x2b00, string) the name of the source file

`DW_AT_ghc_span_start_line` (0x2b01, integer) the line number of the beginning of the span

`DW_AT_ghc_span_start_col` (0x2b02, integer) the column number of the beginning of the span

`DW_AT_ghc_span_end_line` (0x2b03, integer) the line number of the end of the span

`DW_AT_ghc_span_end_col` (0x2b04, integer) the column number of the end of the span

9.5 Further Reading

For more information about the debug information produced by GHC see Peter Wortmann's PhD thesis, *Profiling Optimized Haskell: Causal Analysis and Implementation*.

9.6 Direct Mapping

In addition to the DWARF debug information, which can be used by many standard tools, there is also a GHC specific way to map info table pointers to a source location. This lookup table is generated by using the `-finfo-table-map` flag.

`-finfo-table-map`

> **Since** 9.2

This flag enables the generation of a table which maps the address of an info table to an approximate source position of where that info table statically originated from. If you also want more precise information about constructor info tables then you should also use *`-fdistinct-constructor-tables`* (page 688).

The *`-finfo-table-map`* (page 687) flag will increase the binary size by quite a lot, depending on how big your project is. For compiling a project the size of GHC the overhead was about 200 megabytes.

> **Since** 9.8

If you wish to reduce the size of *`-finfo-table-map`* (page 687) enabled binaries, consider building GHC from source and supplying the `--enable-ipe-data-compression` flag to the `configure` script. This will cause GHC to compress the *`-finfo-table-map`* (page 687) related debugging information included in binaries using the libzstd compression library. **Note**: This feature requires that the machine building GHC has libzstd installed. The compression library `libzstd` may optionally be statically linked in the resulting compiler (on non-darwin machines) using the `--enable-static-libzstd` configure flag.

In a test compiling GHC itself, the size of the *`-finfo-table-map`* (page 687) enabled build results was reduced by over 20% when compression was enabled.

> **Since** 9.10
>
> **Implies** *`-finfo-table-map-with-stack`* (page 687)
>
> **Implies** *`-finfo-table-map-with-fallback`* (page 687)

`-finfo-table-map-with-stack`

> **Since** 9.10

Include info tables for `STACK` closures in the info table map. Note that this flag is implied by *`-finfo-table-map`* (page 687).

`-fno-info-table-map-with-stack`

> **Since** 9.10

`STACK` info tables are often the majority of entries in the info table map. However, despite their contribution to the executable size, they are rarely useful unless debugging with a tool such as ghc-debug. Use this flag to omit `STACK` info tables from the info table map and decrease the size of executables with info table profiling information.

`-finfo-table-map-with-fallback`

> **Since** 9.10

Include info tables with no source location information in the info table map. Note that this flag is implied by *`-finfo-table-map`* (page 687).

`-fno-info-table-map-with-fallback`

Since 9.10

Some info tables, such as those for primitive closure types, will have no provenance location in the program source. With *-finfo-table-map* (page 687), those info tables are given default source locations and included in the info table map. Use this flag to omit them from the info table map and decrease the size of executables with info table profiling information.

-fdistinct-constructor-tables

Since 9.2

For every usage of a data constructor in the source program a new info table will be created. This is useful with *-finfo-table-map* (page 687) and the *-hi* (page 666) profiling mode as each info table will correspond to the usage of a data constructor rather than the data constructor itself.

9.7 Querying the Info Table Map

If it is generated then the info table map can be used in two ways.

1. The `whereFrom` Haskell function can be used to determine the source position which we think a specific closure was created.
2. The complete mapping is also dumped into the eventlog.

If you are using gdb then you can use the `lookupIPE` function (provided by `IPE.h` and exported in the public API) directly in order to find any information which is known about the info table for a specific closure.

CHAPTER

TEN

WHAT TO DO WHEN SOMETHING GOES WRONG

If you still have a problem after consulting this section, then you may have found a *bug*—please report it! See *Reporting bugs in GHC* (page 4) for details on how to report a bug and a list of things we'd like to know about your bug. If in doubt, send a report — we love mail from irate users :-!

(*Haskell standards vs. Glasgow Haskell: language non-compliance* (page 731), which describes Glasgow Haskell's shortcomings vs. the Haskell language definition, may also be of interest.)

10.1 When the compiler "does the wrong thing"

"Help! The compiler crashed (or panic'd)!" These events are *always* bugs in the GHC system—please report them.

"This is a terrible error message." If you think that GHC could have produced a better error message, please report it as a bug.

"What about this warning from the C compiler?" For example: `…warning: \`Foo' declared \`static' but never defined.` Unsightly, but shouldn't be a problem.

Sensitivity to `.hi` interface files GHC is very sensitive about interface files. For example, if it picks up a non-standard `Prelude.hi` file, pretty terrible things will happen.

Furthermore, as sketched below, you may have big problems running programs compiled using unstable interfaces.

"I think GHC is producing incorrect code" Unlikely :-) A useful be-more-paranoid option to give to GHC is `-dcore-lint`-dcore-lint option; this causes a "lint" pass to check for errors (notably type errors) after each Core-to-Core transformation pass. We run with `-dcore-lint` on all the time; it costs about 5% in compile time.

Why did I get a link error? If the linker complains about not finding `_<something>_fast`, then something is inconsistent: you probably didn't compile modules in the proper dependency order.

"Is this line number right?" On this score, GHC usually does pretty well, especially if you "allow" it to be off by one or two. In the case of an instance or class declaration, the line number may only point you to the declaration, not to a specific method.

Please report line-number errors that you find particularly unhelpful.

10.2 When your program "does the wrong thing"

(For advice about overly slow or memory-hungry Haskell programs, please see *Hints* (page 691)).

"Help! My program crashed!" (e.g., a "segmentation fault" or "core dumped") segmentation fault

If your program has no foreign calls in it, and no calls to known-unsafe functions (such as `unsafePerformIO`) then a crash is always a BUG in the GHC system, except in one case: If your program is made of several modules, each module must have been compiled after any modules on which it depends (unless you use `.hi-boot` files, in which case these *must* be correct with respect to the module source).

For example, if an interface is lying about the type of an imported value then GHC may well generate duff code for the importing module. *This applies to pragmas inside interfaces too!* If the pragma is lying (e.g., about the "arity" of a value), then duff code may result. Furthermore, arities may change even if types do not.

In short, if you compile a module and its interface changes, then all the modules that import that interface *must* be re-compiled.

A useful option to alert you when interfaces change is `-ddump-hi-diffs` option. It will run `diff` on the changed interface file, before and after, when applicable.

If you are using `make`, GHC can automatically generate the dependencies required in order to make sure that every module *is* up-to-date with respect to its imported interfaces. Please see *Dependency generation* (page 216).

If you are down to your last-compile-before-a-bug-report, we would recommend that you add a `-dcore-lint` option (for extra checking) to your compilation options.

So, before you report a bug because of a core dump, you should probably:

```
% rm *.o        # scrub your object files
% make my_prog  # re-make your program; use -ddump-hi-diffs to highlight changes;
                # as mentioned above, use -dcore-lint to be more paranoid
% ./my_prog ... # retry...
```

Of course, if you have foreign calls in your program then all bets are off, because you can trash the heap, the stack, or whatever.

"My program entered an 'absent' argument." This is definitely caused by a bug in GHC. Please report it (see *Reporting bugs in GHC* (page 4)).

"What's with this arithmetic (or floating-point) exception?" `Int`, `Float`, and `Double` arithmetic is *unchecked*. Overflows, underflows and loss of precision are either silent or reported as an exception by the operating system (depending on the platform). Divide-by-zero *may* cause an untrapped exception (please report it if it does).

CHAPTER

ELEVEN

HINTS

Please advise us of other "helpful hints" that should go here!

11.1 Sooner: producing a program more quickly

Don't use *-O* (page 113) or (especially) *-O2* (page 113): By using them, you are telling GHC that you are willing to suffer longer compilation times for better-quality code.

GHC is surprisingly zippy for normal compilations without *-O* (page 113)!

Use more memory: Within reason, more memory for heap space means less garbage collection for GHC, which means less compilation time. If you use the `-Rghc-timing` option, you'll get a garbage-collector report. (Again, you can use the cheap-and-nasty `+RTS -S -RTS` option to send the GC stats straight to standard error.)

If it says you're using more than 20% of total time in garbage collecting, then more memory might help: use the `-H⟨size⟩` (see *-H [⟨size⟩]* (page 188)) option. Increasing the default allocation area size used by the compiler's RTS might also help: use the `+RTS -A⟨size⟩ -RTS` option (see *-A ⟨size⟩* (page 185)).

If GHC persists in being a bad memory citizen, please report it as a bug.

Don't use too much memory! As soon as GHC plus its "fellow citizens" (other processes on your machine) start using more than the *real memory* on your machine, and the machine starts "thrashing," *the party is over*. Compile times will be worse than terrible! Use something like the csh builtin **time** command to get a report on how many page faults you're getting.

If you don't know what virtual memory, thrashing, and page faults are, or you don't know the memory configuration of your machine, *don't* try to be clever about memory use: you'll just make your life a misery (and for other people, too, probably).

Try to use local disks when linking: Because Haskell objects and libraries tend to be large, it can take many real seconds to slurp the bits to/from a remote filesystem.

It would be quite sensible to *compile* on a fast machine using remotely-mounted disks; then *link* on a slow machine that had your disks directly mounted.

Don't derive/use `Read` unnecessarily: It's ugly and slow.

GHC compiles some program constructs slowly: We'd rather you reported such behaviour as a bug, so that we can try to correct it.

To figure out which part of the compiler is badly behaved, the `-v2` option is your friend.

11.2 Faster: producing a program that runs quicker

The key tool to use in making your Haskell program run faster are GHC's profiling facilities, described separately in *Profiling* (page 651). There is *no substitute* for finding where your program's time/space is *really* going, as opposed to where you imagine it is going.

Another point to bear in mind: By far the best way to improve a program's performance *dramatically* is to use better algorithms. Once profiling has thrown the spotlight on the guilty time-consumer(s), it may be better to re-think your program than to try all the tweaks listed below.

Another extremely efficient way to make your program snappy is to use library code that has been Seriously Tuned By Someone Else. You *might* be able to write a better quicksort than the one in `Data.List`, but it will take you much longer than typing `import Data.List`.

Please report any overly-slow GHC-compiled programs. Since GHC doesn't have any credible competition in the performance department these days it's hard to say what overly-slow means, so just use your judgement! Of course, if a GHC compiled program runs slower than the same program compiled with NHC or Hugs, then it's definitely a bug.

Optimise, using `-O` or `-O2`: This is the most basic way to make your program go faster. Compilation time will be slower, especially with `-O2`.

At present, `-O2` is nearly indistinguishable from `-O`.

Compile via LLVM: The *LLVM code generator* (page 236) can sometimes do a far better job at producing fast code than the *native code generator* (page 236). This is not universal and depends on the code. Numeric heavy code seems to show the best improvement when compiled via LLVM. You can also experiment with passing specific flags to LLVM with the *`-optlo ⟨option⟩`* (page 239) and *`-optlc ⟨option⟩`* (page 239) flags. Be careful though as setting these flags stops GHC from setting its usual flags for the LLVM optimiser and compiler.

Overloaded functions are not your friend: Haskell's overloading (using type classes) is elegant, neat, etc., etc., but it is death to performance if left to linger in an inner loop. How can you squash it?

Give explicit type signatures: Signatures are the basic trick; putting them on exported, top-level functions is good software-engineering practice, anyway. (Tip: using the *`-Wmissing-signatures`* (page 97) option can help enforce good signature-practice).

The automatic specialisation of overloaded functions (with `-O`) should take care of overloaded local and/or unexported functions.

Use `SPECIALIZE` pragmas: Specialize the overloading on key functions in your program. See *SPECIALIZE pragma* (page 614) and *SPECIALIZE instance pragma* (page 617).

"But how do I know where overloading is creeping in?" A low-tech way: grep (search) your interface files for overloaded type signatures. You can view interface files using the *`--show-iface ⟨file⟩`* (page 69) option (see *Other options related to interface files* (page 205)).

```
$ ghc --show-iface Foo.hi | grep -E '^[a-z].*::.*=>'
```

Strict functions are your dear friends: And, among other things, lazy pattern-matching is your enemy.

(If you don't know what a "strict function" is, please consult a functional-programming textbook. A sentence or two of explanation here probably would not do much good.)

Consider these two code fragments:

```
f (Wibble x y) =  ... # strict

f arg = let { (Wibble x y) = arg } in ... # lazy
```

The former will result in far better code.

A less contrived example shows the use of `BangPatterns` on `lets` to get stricter code (a good thing):

```
f (Wibble x y)
      = let
            !(a1, b1, c1) = unpackFoo x
            !(a2, b2, c2) = unpackFoo y
        in ...
```

GHC loves single-constructor data-types: It's all the better if a function is strict in a single-constructor type (a type with only one data-constructor; for example, tuples are single-constructor types).

Newtypes are better than datatypes: If your datatype has a single constructor with a single field, use a `newtype` declaration instead of a `data` declaration. The `newtype` will be optimised away in most cases.

"How do I find out a function's strictness?" Don't guess—look it up.

Look for your function in the interface file, then for the third field in the pragma; it should say `Strictness: ⟨string⟩`. The ⟨string⟩ gives the strictness of the function's arguments: see the GHC Commentary for a description of the strictness notation.

For an "unpackable" `U(...)` argument, the info inside tells the strictness of its components. So, if the argument is a pair, and it says `U(AU(LSS))`, that means "the first component of the pair isn't used; the second component is itself unpackable, with three components (lazy in the first, strict in the second \& third)."

If the function isn't exported, just compile with the extra flag *-ddump-simpl* (page 259); next to the signature for any binder, it will print the self-same pragmatic information as would be put in an interface file. (Besides, Core syntax is fun to look at!)

Force key functions to be `INLINE`d (esp. monads): Placing `INLINE` pragmas on certain functions that are used a lot can have a dramatic effect. See *INLINE pragma* (page 609).

Explicit `export` list: If you do not have an explicit export list in a module, GHC must assume that everything in that module will be exported. This has various pessimising effects. For example, if a bit of code is actually *unused* (perhaps because of unfolding effects), GHC will not be able to throw it away, because it is exported and some other module may be relying on its existence.

GHC can be quite a bit more aggressive with pieces of code if it knows they are not exported.

Look at the Core syntax! (The form in which GHC manipulates your code.) Just run your compilation with *-ddump-simpl* (page 259) (don't forget the *-O* (page 113)).

If profiling has pointed the finger at particular functions, look at their Core code. `lets` are bad, `cases` are good, dictionaries (`d.⟨Class⟩.⟨Unique⟩`) [or anything overloading-ish] are bad, nested lambdas are bad, explicit data constructors are good, primitive operations (e.g., ==#) are good, …

Use strictness annotations: Putting a strictness annotation (`!`) on a constructor field helps in two ways: it adds strictness to the program, which gives the strictness analyser more to work with, and it might help to reduce space leaks.

It can also help in a third way: when used with *`-funbox-strict-fields`* (page 129) (see *-f*: platform-independent flags* (page 113)), a strict field can be unpacked or unboxed in the constructor, and one or more levels of indirection may be removed. Unpacking only happens for single-constructor datatypes (`Int` is a good candidate, for example).

Using *`-funbox-strict-fields`* (page 129) is only really a good idea in conjunction with *`-O`* (page 113), because otherwise the extra packing and unpacking won't be optimised away. In fact, it is possible that *`-funbox-strict-fields`* (page 129) may worsen performance even *with* *`-O`* (page 113), but this is unlikely (let us know if it happens to you).

Use unboxed types (a GHC extension): When you are *really* desperate for speed, and you want to get right down to the "raw bits." Please see *Unboxed types* (page 554) for some information about using unboxed types.

Before resorting to explicit unboxed types, try using strict constructor fields and *`-funbox-strict-fields`* (page 129) first (see above). That way, your code stays portable.

Use `foreign import` (a GHC extension) to plug into fast libraries: This may take real work, but… There exist piles of massively-tuned library code, and the best thing is not to compete with it, but link with it.

Foreign function interface (FFI) (page 561) describes the foreign function interface.

Don't use `Floats`: If you're using `Complex`, definitely use `Complex Double` rather than `Complex Float` (the former is specialised heavily, but the latter isn't).

`Floats` (probably 32-bits) are almost always a bad idea, anyway, unless you Really Know What You Are Doing. Use `Doubles`. There's rarely a speed disadvantage—modern machines will use the same floating-point unit for both. With `Doubles`, you are much less likely to hang yourself with numerical errors.

One time when `Float` might be a good idea is if you have a *lot* of them, say a giant array of `Floats`. They take up half the space in the heap compared to `Doubles`. However, this isn't true on a 64-bit machine.

Use unboxed arrays (`UArray`) GHC supports arrays of unboxed elements, for several basic arithmetic element types including `Int` and `Char`: see the Data.Array.Unboxed library for details. These arrays are likely to be much faster than using standard Haskell 98 arrays from the Data.Array library.

Use a bigger heap! If your program's GC stats (*`-S [⟨file⟩]`* (page 191) RTS option) indicate that it's doing lots of garbage-collection (say, more than 20% of execution time), more memory might help — with the *`-H [⟨size⟩]`* (page 188) or *`-A ⟨size⟩`* (page 185) RTS options (see *RTS options to control the garbage collector* (page 184)). As a rule of thumb, try setting *`-H [⟨size⟩]`* (page 188) to the amount of memory you're willing to let your process consume, or perhaps try passing *`-H [⟨size⟩]`* (page 188) without any argument to let GHC calculate a value based on the amount of live data.

Compact your data: The GHC.Compact module provides a way to make garbage collection more efficient for long-lived data structures. Compacting a data structure collects the objects together in memory, where they are treated as a single object by the garbage collector and not traversed individually.

11.3 Smaller: producing a program that is smaller

Decrease the "go-for-it" threshold for unfolding smallish expressions. Give a *-funfolding-use-threshold=0* (page 129) option for the extreme case. ("Only unfoldings with zero cost should proceed.") Warning: except in certain specialised cases (like Happy parsers) this is likely to actually *increase* the size of your program, because unfolding generally enables extra simplifying optimisations to be performed.

Avoid Prelude.Read.

Use **strip** on your executables.

11.4 Thriftier: producing a program that gobbles less heap space

"I think I have a space leak…"

Re-run your program with *+RTS -S* (page 191), and remove all doubt! (You'll see the heap usage get bigger and bigger…) (Hmmm… this might be even easier with the *-G1* (page 187) RTS option; so… `./a.out +RTS -S -G1`)

Once again, the profiling facilities (*Profiling* (page 651)) are the basic tool for demystifying the space behaviour of your program.

Strict functions are good for space usage, as they are for time, as discussed in the previous section. Strict functions get right down to business, rather than filling up the heap with closures (the system's notes to itself about how to evaluate something, should it eventually be required).

11.5 Controlling inlining via optimisation flags.

Inlining is one of the major optimizations GHC performs. Partially because inlining often allows other optimizations to be triggered. Sadly this is also a double edged sword. While inlining can often cut through runtime overheads this usually comes at the cost of not just program size, but also compiler performance. In extreme cases making it impossible to compile certain code.

For this reason GHC offers various ways to tune inlining behaviour.

11.5.1 Unfolding creation

In order for a function from a different module to be inlined GHC requires the functions unfolding. The following flags can be used to control unfolding creation. Making their creation more or less likely:

- *-fexpose-all-unfoldings* (page 117)
- *-funfolding-creation-threshold=⟨n⟩* (page 129)

11.5.2 Inlining decisions

If a unfolding is available the following flags can impact GHC's decision about inlining a specific binding.

- *-funfolding-use-threshold=⟨n⟩* (page 129)
- *-funfolding-case-threshold=⟨n⟩* (page 129)
- *-funfolding-case-scaling=⟨n⟩* (page 130)
- *-funfolding-dict-discount=⟨n⟩* (page 129)
- *-funfolding-fun-discount=⟨n⟩* (page 129)

Should the simplifier run out of ticks because of a inlining loop users are encouraged to try decreasing *-funfolding-case-threshold=⟨n⟩* (page 129) or *-funfolding-case-scaling=⟨n⟩* (page 130) to limit inlining into deeply nested expressions while allowing a higher tick factor.

The defaults of these are tuned such that we don't expect regressions for most user programs. Using a *-funfolding-case-threshold=⟨n⟩* (page 129) of 1-2 with a *-funfolding-case-scaling=⟨n⟩* (page 130) of 15-25 can cause usually small runtime regressions but will prevent most inlining loops from getting out of control.

In extreme cases lowering scaling and threshold further can be useful, but at that point it's very likely that beneficial inlining is prevented as well resulting in significant runtime regressions.

In such cases it's recommended to move the problematic piece of code into it's own module and changing inline parameters for the offending module only.

11.5.3 Inlining generics

There are also flags specific to the inlining of generics:

- *-finline-generics* (page 124)
- *-finline-generics-aggressively* (page 125)

11.6 Understanding how OS memory usage corresponds to live data

A confusing aspect about the RTS is the sometimes big difference between OS reported memory usage and the amount of live data reported by heap profiling or `GHC.Stats`.

There are two main factors which determine OS memory usage.

Firstly the collection strategy used by the oldest generation. By default a copying strategy is used which requires at least 2 times the amount of currently live data in order to perform a major collection. For example, if your program's live data is 1G then you would expect the OS to report at minimum 2G.

If instead you are using the compacting (*-c* (page 186)) or nonmoving (*-xn* (page 184)) strategies for the oldest generation then less overhead is required as the strategy immediately reuses already allocated memory by overwriting. For a program with heap size 1G then you might expect the OS to report at minimum a small percentage above 1G.

Secondly, after doing some allocation GHC is quite reluctant to return the memory to the OS. This is because after performing a major collection the program might still be allocating a lot and it costs to have to request more memory. Therefore the RTS keeps an extra amount to reuse which depends on the *-F ⟨factor⟩* (page 186) option. By default the RTS will keep up to `(2 + F) * live_bytes` after performing a major collection due to exhausting the available heap. The default value is `F = 2` so you can see OS memory usage reported to be as high as 4 times the amount used by your program.

Without further intervention, once your program has topped out at this high threshold, no more memory would be returned to the OS so memory usage would always remain at 4 times the live data. If you had a server with 1.5G live data, then if there was a memory spike up to 6G for a short period, then OS reported memory would never dip below 6G. This is what happened before GHC 9.2. In GHC 9.2 memory is gradually returned to the OS so OS memory usage returns closer to the theoretical minimums.

The *-Fd ⟨factor⟩* (page 186) option controls the rate at which memory is returned to the OS. On consecutive major collections which are not triggered by heap overflows, a counter (`t`) is increased and the `F` factor is inversly scaled according to the value of `t` and `Fd`. The factor is scaled by the equation:

$$\mathsf{F}' = \mathsf{F} \times 2^{\frac{-\mathsf{t}}{\mathsf{Fd}}}$$

By default `Fd = 4`, increasing `Fd` decreases the rate memory is returned.

Major collections which are not triggered by heap overflows arise mainly in two ways.

1. Idle collections (controlled by *-I ⟨seconds⟩* (page 188))
2. Explicit trigger using `performMajorGC`.

For example, idle collections happen by default after 0.3 seconds of inactivity. If you are running your application and have also set `-Iw30`, so that the minimum period between idle GCs is 30 seconds, then say you do a small amount of work every 5 seconds, there will be about 10 idle collections about 5 minutes. This number of consecutive idle collections will scale the `F` factor as follows:

$$\mathsf{F}' = 2 \times 2^{\frac{-10}{4}} \approx 0.35$$

and hence we will only retain `(0.35 + 2) * live_bytes` rather than the original 4 times. If you want less frequent idle collections then you should also decrease `Fd` so that more memory is returned each time a collection takes place.

If you set `-Fd0` then GHC will not attempt to return memory, which corresponds with the behaviour from releases prior to 9.2. You probably don't want to do this as unless you have idle periods in your program the behaviour will be similar anyway. If you want to retain a specific amount of memory then it's better to set `-H1G` in order to communicate that you are happy with a heap size of `1G`. If you do this then OS reported memory will never decrease below this amount if it ever reaches this threshold.

The collecting strategy also affects the fragmentation of the heap and hence how easy it is to return memory to a theoretical baseline. Memory is allocated firstly in the unit of megablocks which is then further divided into blocks. Block-level fragmentation is how much unused space within the allocated megablocks there is. In a fragmented heap there will be many megablocks which are only partially full.

In theory the compacting strategy has a lower memory baseline but practically it can be hard to reach the baseline due to how compacting never defragments. On the other hand, the copying collecting has a higher theoretical baseline but we can often get very close to it because the act of copying leads to lower fragmentation.

There are some other flags which affect the amount of retained memory as well. Setting the maximum heap size using *-M ⟨size⟩* (page 190) will make sure we don't try and retain more memory than the maximum size and explicitly setting *-H [⟨size⟩]* (page 188) will mean that we will always try and retain at least `H` bytes irrespective of the amount of live data.

CHAPTER

TWELVE

OTHER HASKELL UTILITY PROGRAMS

This section describes other program(s) which we distribute, that help with the Great Haskell Programming Task.

12.1 "Yacc for Haskell": `happy`

Andy Gill and Simon Marlow have written a parser-generator for Haskell, called `happy`. `Happy` is to Haskell what `Yacc` is to C.

You can get `happy` from the Happy Homepage.

`Happy` is at its shining best when compiled by GHC.

12.2 Writing Haskell interfaces to C code: `hsc2hs`

The `hsc2hs` command can be used to automate some parts of the process of writing Haskell bindings to C code. It reads an almost-Haskell source with embedded special constructs, and outputs a real Haskell file with these constructs processed, based on information taken from some C headers. The extra constructs deal with accessing C data from Haskell.

It may also output a C file which contains additional C functions to be linked into the program, together with a C header that gets included into the C code to which the Haskell module will be compiled (when compiled via C) and into the C file. These two files are created when the `#def` construct is used (see below).

Actually `hsc2hs` does not output the Haskell file directly. It creates a C program that includes the headers, gets automatically compiled and run. That program outputs the Haskell code.

In the following, "Haskell file" is the main output (usually a `.hs` file), "compiled Haskell file" is the Haskell file after `ghc` has compiled it to C (i.e. a `.hc` file), "C program" is the program that outputs the Haskell file, "C file" is the optionally generated C file, and "C header" is its header file.

12.2.1 command line syntax

`hsc2hs` takes input files as arguments, and flags that modify its behavior:

-o FILE, --output=FILE Name of the Haskell file.

-t FILE, --template=FILE The template file (see below).

-c PROG, --cc=PROG The C compiler to use (default: `gcc`)

-l PROG, --ld=PROG The linker to use (default: `gcc`).

-C FLAG, --cflag=FLAG An extra flag to pass to the C compiler.

-I DIR Passed to the C compiler.

-L FLAG, --lflag=FLAG An extra flag to pass to the linker.

-i FILE, --include=FILE As if the appropriate `#include` directive was placed in the source.

-D NAME[=VALUE], --define=NAME[=VALUE] As if the appropriate `#define` directive was placed in the source.

--no-compile Stop after writing out the intermediate C program to disk. The file name for the intermediate C program is the input file name with `.hsc` replaced with `_hsc_make.c`.

-k, --keep-files Proceed as normal, but do not delete any intermediate files.

-x, --cross-compile Activate cross-compilation mode (see *Cross-compilation* (page 702)).

--cross-safe Restrict the .hsc directives to those supported by the `--cross-compile` mode (see *Cross-compilation* (page 702)). This should be useful if your `.hsc` files must be safely cross-compiled and you wish to keep non-cross-compilable constructs from creeping into them.

-?, --help Display a summary of the available flags and exit successfully.

-V, --version Output version information and exit successfully.

The input file should end with .hsc (it should be plain Haskell source only; literate Haskell is not supported at the moment). Output files by default get names with the `.hsc` suffix replaced:

`.hs`	Haskell file
`_hsc.h`	C header
`_hsc.c`	C file

The C program is compiled using the Haskell compiler. This provides the include path to `HsFFI.h` which is automatically included into the C program.

12.2.2 Input syntax

All special processing is triggered by the # operator. To output a literal #, write it twice: ##. Inside string literals and comments # characters are not processed.

A # is followed by optional spaces and tabs, an alphanumeric keyword that describes the kind of processing, and its arguments. Arguments look like C expressions separated by commas (they are not written inside parens). They extend up to the nearest unmatched `)`, `]` or `}`, or to the end of line if it occurs outside any `() [] {} '' "" /**/` and is not preceded by a backslash. Backslash-newline pairs are stripped.

In addition #{stuff} is equivalent to #stuff except that it's self-delimited and thus needs not to be placed at the end of line or in some brackets.

Meanings of specific keywords:

#include <file.h>, #include "file.h" The specified file gets included into the C program, the compiled Haskell file, and the C header. <HsFFI.h> is included automatically.

#define ⟨name⟩, #define ⟨name ⟨value⟩, #undef ⟨name⟩ Similar to #include. Note that #includes and #defines may be put in the same file twice so they should not assume otherwise.

#let ⟨name⟩ ⟨parameters⟩ = "⟨definition⟩" Defines a macro to be applied to the Haskell source. Parameter names are comma-separated, not inside parens. Such macro is invoked as other #-constructs, starting with #name. The definition will be put in the C program inside parens as arguments of printf. To refer to a parameter, close the quote, put a parameter name and open the quote again, to let C string literals concatenate. Or use printf's format directives. Values of arguments must be given as strings, unless the macro stringifies them itself using the C preprocessor's #parameter syntax.

#def ⟨C_definition⟩ The definition (of a function, variable, struct or typedef) is written to the C file, and its prototype or extern declaration to the C header. Inline functions are handled correctly. struct definitions and typedefs are written to the C program too. The inline, struct or typedef keyword must come just after def.

#if ⟨condition⟩, #ifdef ⟨name⟩, #ifndef ⟨name⟩, #elif ⟨condition⟩, #else, #endif, #error ⟨messa
Conditional compilation directives are passed unmodified to the C program, C file, and C header. Putting them in the C program means that appropriate parts of the Haskell file will be skipped.

#const ⟨C_expression⟩ The expression must be convertible to long or unsigned long. Its value (literal or negated literal) will be output.

#const_str ⟨C_expression⟩ The expression must be convertible to const char pointer. Its value (string literal) will be output.

#type ⟨C_type⟩ A Haskell equivalent of the C numeric type will be output. It will be one of {Int,Word}{8,16,32,64}, Float, Double, LDouble.

#peek ⟨struct_type⟩, ⟨field⟩ A function that peeks a field of a C struct will be output. It will have the type Storable b => Ptr a -> IO b. The intention is that #peek and #poke can be used for implementing the operations of class Storable for a given C struct (see the Foreign.Storable module in the library documentation).

#poke ⟨struct_type⟩, ⟨field⟩ Similarly for poke. It will have the type Storable b => Ptr a -> b -> IO ().

#ptr ⟨struct_type⟩, ⟨field⟩ Makes a pointer to a field struct. It will have the type Ptr a -> Ptr b.

#offset ⟨struct_type⟩, ⟨field⟩ Computes the offset, in bytes, of field in struct_type. It will have type Int.

#size ⟨struct_type⟩ Computes the size, in bytes, of struct_type. It will have type Int.

#alignment ⟨struct_type⟩ Computes the alignment, in bytes, of struct_type. It will have type Int.

#enum ⟨type⟩, ⟨constructor⟩, ⟨value⟩, ⟨value⟩, ... A shortcut for multiple definitions which use #const. Each value is a name of a C integer constant, e.g. enumeration value. The name will be translated to Haskell by making each letter following an underscore uppercase, making all the rest lowercase, and removing underscores. You can supply a different translation by writing `hs_name = c_value` instead of a `value`, in which case `c_value` may be an arbitrary expression. The `hs_name` will be defined as having the specified `type`. Its definition is the specified `constructor` (which in fact may be an expression or be empty) applied to the appropriate integer value. You can have multiple `#enum` definitions with the same `type`; this construct does not emit the type definition itself.

12.2.3 Custom constructs

`#const`, `#type`, `#peek`, `#poke` and `#ptr` are not hardwired into the `hsc2hs`, but are defined in a C template that is included in the C program: `template-hsc.h`. Custom constructs and templates can be used too. Any #-construct with unknown key is expected to be handled by a C template.

A C template should define a macro or function with name prefixed by `hsc_` that handles the construct by emitting the expansion to stdout. See `template-hsc.h` for examples.

Such macros can also be defined directly in the source. They are useful for making a `#let`-like macro whose expansion uses other `#let` macros. Plain `#let` prepends `hsc_` to the macro name and wraps the definition in a `printf` call.

12.2.4 Cross-compilation

`hsc2hs` normally operates by creating, compiling, and running a C program. That approach doesn't work when cross-compiling — in this case, the C compiler's generates code for the target machine, not the host machine. For this situation, there's a special mode `hsc2hs --cross-compile` which can generate the .hs by extracting information from compilations only — specifically, whether or not compilation fails.

Only a subset of `.hsc` syntax is supported by `--cross-compile`. The following are unsupported:

- `#{const_str}`
- `#{let}`
- `#{def}`
- Custom constructs

CHAPTER

THIRTEEN

RUNNING GHC ON WIN32 SYSTEMS

13.1 Starting GHC on Windows platforms

The installer that installs GHC on Win32 also sets up the file-suffix associations for ".hs" and ".lhs" files so that double-clicking them starts `ghci`.

Be aware of that `ghc` and `ghci` do require filenames containing spaces to be escaped using quotes:

```
c:\ghc\bin\ghci "c:\\Program Files\\Haskell\\Project.hs"
```

If the quotes are left off in the above command, `ghci` will interpret the filename as two, `c:\\\\Program` and `Files\\\\Haskell\\\\Project.hs`.

13.2 Running GHCi on Windows

We recommend running GHCi in a standard Windows console: select the `GHCi` option from the start menu item added by the GHC installer, or use `Start->Run->cmd` to get a Windows console and invoke `ghci` from there (as long as it's in your `PATH`).

If you run GHCi in a Cygwin or MSYS shell, then the Control-C behaviour is adversely affected. In one of these environments you should use the `ghcii.sh` script to start GHCi, otherwise when you hit Control-C you'll be returned to the shell prompt but the GHCi process will still be running. However, even using the `ghcii.sh` script, if you hit Control-C then the GHCi process will be killed immediately, rather than letting you interrupt a running program inside GHCi as it should. This problem is caused by the fact that the Cygwin and MSYS shell environments don't pass Control-C events to non-Cygwin child processes, because in order to do that there needs to be a Windows console.

There's an exception: you can use a Cygwin shell if the `CYGWIN` environment variable does *not* contain `tty`. In this mode, the Cygwin shell behaves like a Windows console shell and console events are propagated to child processes. Note that the `CYGWIN` environment variable must be set *before* starting the Cygwin shell; changing it afterwards has no effect on the shell.

This problem doesn't just affect GHCi, it affects any GHC-compiled program that wants to catch console events. See the GHC.ConsoleHandler module.

13.3 Interacting with the terminal

By default GHC builds applications that open a console window when they start. If you want to build a GUI-only application, with no console window, use the flag `-optl-mwindows` in the link step.

> **Warning:** Windows GUI-only programs have no stdin, stdout or stderr so using the ordinary Haskell input/output functions will cause your program to fail with an IO exception, such as:
>
> ```
> Fail: <stdout>: hPutChar: failed (Bad file descriptor)
> ```
>
> However using Debug.Trace.trace is alright because it uses Windows debugging output support rather than `stderr`.

For some reason, Mingw ships with the `readline` library, but not with the `readline` headers. As a result, GHC (like Hugs) does not use `readline` for interactive input on Windows. You can get a close simulation by using an emacs shell buffer!

13.4 Differences in library behaviour

Some of the standard Haskell libraries behave slightly differently on Windows.

- On Windows, the `^Z` character is interpreted as an end-of-file character, so if you read a file containing this character the file will appear to end just before it. To avoid this, use `IOExts.openFileEx` to open a file in binary (untranslated) mode or change an already opened file handle into binary mode using `IOExts.hSetBinaryMode`. The `IOExts` module is part of the `lang` package.

13.5 File paths under Windows

Windows paths are not all the same. The different kinds of paths each have different meanings. The `MAX_PATH` limitation is not a limitation of the operating system nor the file system. It is a limitation of the default namespace enforced by the Win32 API for backwards compatibility.

The NT kernel however allows you ways to opt out of this path preprocessing by the Win32 APIs. This is done by explicitly using the desired namespace in the path.

The namespaces are:

- file namespace: `\\?\`
- device namespace: `\\.\`
- NT namespace: `\`

Each of these turn off path processing completely by the Win32 API and the paths are passed untouched to the filesystem.

Paths with a drive letter are *legacy* paths. The drive letters are actually meaningless to the kernel. Just like Unix operating systems, drive letters are just a mount point. You can view your mount points by using the **`mountvol`** command.

Since GHC 8.6.1, the Haskell I/O manager automatically promotes paths in the legacy format to Win32 file namespace. By default the I/O manager will do two things to your paths:

- replace `\` with `\\`
- expand relative paths to absolute paths

If you want to opt out of all preprocessing just explicitly use namespaces in your paths. Due to this change, if you need to open raw devices (e.g. COM ports) you need to use the device namespace explicitly. (e.g. `\\.\COM1`). GHC and Haskell programs in general no longer support opening devices in the legacy format.

See the Windows documentation for more details.

13.6 Using GHC (and other GHC-compiled executables) with Cygwin

13.6.1 Background

The Cygwin tools aim to provide a Unix-style API on top of the windows libraries, to facilitate ports of Unix software to windows. To this end, they introduce a Unix-style directory hierarchy under some root directory (typically `/` is `C:\cygwin\`). Moreover, everything built against the Cygwin API (including the Cygwin tools and programs compiled with Cygwin's GHC) will see `/` as the root of their file system, happily pretending to work in a typical unix environment, and finding things like `/bin` and `/usr/include` without ever explicitly bothering with their actual location on the windows system (probably `C:\cygwin\bin` and `C:\cygwin\usr\include`).

13.6.2 The problem

GHC, by default, no longer depends on cygwin, but is a native Windows program. It is built using mingw, and it uses mingw's GHC while compiling your Haskell sources (even if you call it from cygwin's bash), but what matters here is that - just like any other normal windows program - neither GHC nor the executables it produces are aware of Cygwin's pretended unix hierarchy. GHC will happily accept either `/` or `\\` as path separators, but it won't know where to find `/home/joe/Main.hs` or `/bin/bash` or the like. This causes all kinds of fun when GHC is used from within Cygwin's bash, or in make-sessions running under Cygwin.

13.6.3 Things to do

- Don't use absolute paths in `make`, `configure` & co if there is any chance that those might be passed to GHC (or to GHC-compiled programs). Relative paths are fine because cygwin tools are happy with them and GHC accepts `/` as path-separator. And relative paths don't depend on where Cygwin's root directory is located, or on which partition or network drive your source tree happens to reside, as long as you `cd` there first.
- If you have to use absolute paths (beware of the innocent-looking `ROOT=$(pwd)` in makefile hierarchies or configure scripts), Cygwin provides a tool called `cygpath` that can convert Cygwin's Unix-style paths to their actual Windows-style counterparts. Many Cygwin tools actually accept absolute Windows-style paths (remember, though, that you either need to escape `\\` or convert `\\` to `/`), so you should be fine just using those everywhere. If you need to use tools that do some kind of path-mangling that depends on unix-style paths (one fun example is trying to interpret `:` as a separator in path lists),

you can still try to convert paths using `cygpath` just before they are passed to GHC and friends.

- If you don't have `cygpath`, you probably don't have cygwin and hence no problems with it... unless you want to write one build process for several platforms. Again, relative paths are your friend, but if you have to use absolute paths, and don't want to use different tools on different platforms, you can simply write a short Haskell program to print the current directory (thanks to George Russell for this idea): compiled with GHC, this will give you the view of the file system that GHC depends on (which will differ depending on whether GHC is compiled with cygwin's gcc or mingw's gcc or on a real Unix system..) - that little program can also deal with escaping \\ in paths. Apart from the banner and the startup time, something like this would also do:

```
$ echo "Directory.getCurrentDirectory >>= putStrLn . init . tail . show " | ghci
```

13.7 Building and using Win32 DLLs

Dynamic link libraries, Win32 DLLs, Win32 On Win32 platforms, the compiler is capable of both producing and using dynamic link libraries (DLLs) containing ghc-compiled code. This section shows you how to make use of this facility.

There are two distinct ways in which DLLs can be used:

- You can turn each Haskell package into a DLL, so that multiple Haskell executables using the same packages can share the DLL files. (As opposed to linking the libraries statically, which in effect creates a new copy of the RTS and all libraries for each executable produced.)

  That is the same as the dynamic linking on other platforms, and it is described in *Using shared libraries* (page 251).

- You can package up a complete Haskell program as a DLL, to be called by some external (usually non-Haskell) program. This is usually used to implement plugins and the like, and is described below.

13.7.1 Creating a DLL

Creating a Win32 DLL -shared Sealing up your Haskell library inside a DLL is straightforward; compile up the object files that make up the library, and then build the DLL by issuing a command of the form:

```
ghc -shared -o foo.dll bar.o baz.o wibble.a -lfooble
```

By feeding the ghc compiler driver the option `-shared`, it will build a DLL rather than produce an executable. The DLL will consist of all the object files and archives given on the command line.

A couple of things to notice:

- By default, the entry points of all the object files will be exported from the DLL when using `-shared`. Should you want to constrain this, you can specify the *module definition file* to use on the command line as follows:

  ```
  ghc -shared -o .... MyDef.def
  ```

See Microsoft documentation for details, but a module definition file simply lists what entry points you want to export. Here's one that's suitable when building a Haskell COM server DLL:

```
EXPORTS
 DllCanUnloadNow     = DllCanUnloadNow@0
 DllGetClassObject   = DllGetClassObject@12
 DllRegisterServer   = DllRegisterServer@0
 DllUnregisterServer = DllUnregisterServer@0
```

- In addition to creating a DLL, the `-shared` option also creates an import library. The import library name is derived from the name of the DLL, as follows:

```
DLL: HScool.dll  ==> import lib: libHScool.dll.a
```

The naming scheme may look a bit weird, but it has the purpose of allowing the co-existence of import libraries with ordinary static libraries (e.g., `libHSfoo.a` and `libHSfoo.dll.a`. Additionally, when the compiler driver is linking in non-static mode, it will rewrite occurrence of `-lHSfoo` on the command line to `-lHSfoo.dll`. By doing this for you, switching from non-static to static linking is simply a question of adding `-static` to your command line.

13.7.2 Making DLLs to be called from other languages

This section describes how to create DLLs to be called from other languages, such as Visual Basic or C++. This is a special case of *Making a Haskell library that can be called from foreign code* (page 571); we'll deal with the DLL-specific issues that arise below. Here's an example:

Use foreign export declarations to export the Haskell functions you want to call from the outside. For example:

```haskell
-- Adder.hs
{-# LANGUAGE ForeignFunctionInterface #-}
module Adder where

adder :: Int -> Int -> IO Int  -- gratuitous use of IO
adder x y = return (x+y)

foreign export stdcall adder :: Int -> Int -> IO Int
```

Add some helper code that starts up and shuts down the Haskell RTS:

```c
// StartEnd.c
#include <Rts.h>

void HsStart()
{
   int argc = 1;
   char* argv[] = {"ghcDll", NULL}; // argv must end with NULL

   // Initialize Haskell runtime
   char** args = argv;
   hs_init(&argc, &args);
}

```

(continues on next page)

(continued from previous page)

```
void HsEnd()
{
   hs_exit();
}
```

Here, `Adder` is the name of the root module in the module tree (as mentioned above, there must be a single root module, and hence a single module tree in the DLL). Compile everything up:

```
ghc -c Adder.hs
ghc -c StartEnd.c
ghc -shared -o Adder.dll Adder.o Adder_stub.o StartEnd.o
```

Now the file `Adder.dll` can be used from other programming languages. Before calling any functions in Adder it is necessary to call `HsStart`, and at the very end call `HsEnd`.

> **Warning:** It may appear tempting to use `DllMain` to call `hs_init`/`hs_exit`, but this won't work (particularly if you compile with `-threaded`). There are severe restrictions on which actions can be performed during `DllMain`, and `hs_init` violates these restrictions, which can lead to your DLL freezing during startup (see #3605).

13.7.2.1 Using from VBA

An example of using `Adder.dll` from VBA is:

```
Private Declare Function Adder Lib "Adder.dll" Alias "adder@8" _
      (ByVal x As Long, ByVal y As Long) As Long

Private Declare Sub HsStart Lib "Adder.dll" ()
Private Declare Sub HsEnd Lib "Adder.dll" ()

Private Sub Document_Close()
HsEnd
End Sub

Private Sub Document_Open()
HsStart
End Sub

Public Sub Test()
MsgBox "12 + 5 = " & Adder(12, 5)
End Sub
```

This example uses the `Document_Open`/`Close` functions of Microsoft Word, but provided `HsStart` is called before the first function, and `HsEnd` after the last, then it will work fine.

13.7.2.2 Using from C++

An example of using `Adder.dll` from C++ is:

```
// Tester.cpp
#include "HsFFI.h"
#include "Adder_stub.h"
#include <stdio.h>

extern "C" {
    void HsStart();
    void HsEnd();
}

int main()
{
    HsStart();
    // can now safely call functions from the DLL
    printf("12 + 5 = %i\n", adder(12,5))    ;
    HsEnd();
    return 0;
}
```

This can be compiled and run with:

```
$ ghc -o tester Tester.cpp Adder.dll.a
$ tester
12 + 5 = 17
```

CHAPTER

FOURTEEN

FFI AND THE JAVASCRIPT BACKEND

GHC's JavaScript backend supports its own calling convention for JavaScript-specific foreign imports. Any unapplied function is supported, including function names. Commonly, JavaScript foreign imports are written as an unapplied JavaScript arrow function, but `function` keyword anonymous functions are also supported.

By treating an import string as an unapplied function, arbitrary JavaScript can be included in an import, so a simple example might look like:

```
foreign import javascript "((x,y) => { return x + y; })"
  js_add :: Int -> Int -> Int
```

14.1 JavaScript FFI Types

Some types are able to be used directly in the type signatures of foreign exports, without conversion to a `JSVal`. We saw in the first example that `Int` is one of these.

There are a number of supported types that can be passed directly in this way, and they act as primitives within GHC's JavaScript RTS. This is in comparison to data structures that are implemented in Haskell, such as `String` - being a list, this doesn't have a primitive JavaScript implementation, and isn't equivalent to a JavaScript string.

The following types are supported in this way:

- `Int`, including `Int32` and other sized numerical values.
- `Int64`, and other 64 bit numbers are passed as two variables to the function, where the first includes the sign and the higher bits
- `Bool`
- `Char`
- `Any`
- `ByteArray#`
- `Double` and `Float`
- `MVar#`, and other RTS objects
- Unboxed tuples (e.g. `(# a, b #)`) can appear in the return type, and are constructed in JavaScript using macros such as `RETURN_UBX_TUP2(x, y)`.

As in the C FFI, types in the JavaScript FFI can't be type checked against the foreign code, so the following example would compile successfully - despite *5* not being a valid JavaScript value for the Haskell *Bool* type:

```
foreign import javascript "((x) => { return 5; })"
  type_error :: Bool -> Bool
```

14.1.1 JSVal

The JavaScript backend has a concept of an untyped 'plain' JavaScript value, under the guise of the type `JSVal`. Values having this type are mostly opaque to Haskell codes: you can think of *JSVal* as a data type whose data constructors aren't exposed. Its main use case is to pass opaque JavaScript values from one FFI call to another.

Nevertheless the module `GHC.JS.Prim` from `base` contains functions for working with foreign `JSVal` objects. Currently, it provides the following conversions:

- `Int` <-> `JSVal` (`toJSInt`, `fromJSInt`)
- `String` <-> `JSVal` (`toJSString`, `fromJSString`)
- `[JSVal]` <-> `JSVal` (`toJSArray`, `fromJSArray`)

It also contains functions for working with objects:

- `jsNull :: JSVal` - the JavaScript `null`
- `isNull :: JSVal -> Bool` - test for the JavaScript `null`
- `isUndefined :: JSVal -> Bool` - test for the JavaScript `undefined`
- `getProp :: JSVal -> String -> JSVal` - object field access

14.1.2 JavaScript Callbacks

The JavaScript execution model is based around callback functions, and GHC's JavaScript backend implements these as a type in order to support useful browser programs, and programs interacting with JavaScript libraries.

The module `GHC.JS.Foreign.Callback` in `base` defines the type `Callback a`, as well as several functions to construct callbacks from Haskell functions of up to three `JSVal` arguments. Unlike a regular function, a `Callback` function is passed in the FFI as a plain JavaScript function - enabling us to call these functions from within JavaScript:

```
foreign import javascript "((f) => { f('Example!'); })"
  callback_example :: Callback (JSVal -> IO ()) -> IO ()

printJSValAsString :: JSVal -> IO ()
printJSValAsString = putStrLn . fromJSString

main :: IO ()
main = do
  printJS <- syncCallback1 ThrowWouldBlock printJSValAsString
  callback_example printJS
  releaseCallback printJS
```

This example will call our `printJSValAsString` function, via JavaScript, with the JavaScript string `Example!` as an argument. On the last line, the callback memory is freed. Since there's no way for the Haskell JS runtime to know if a function is still being referenced by JavaScript code, the memory must be manually released when no longer needed.

On the first line of `main`, we see where the `Callback` is actually created, by `syncCallback1`. `syncCallback` has versions up to three, including a zero-argument version with no suffix. To use callbacks with more than three pieces of data, it's recommended to package data into JavaScript objects or arrays as required.

There are three categories of functions that create callbacks, with the arity-1 type signatures shown here for demonstration:

- `syncCallback1 :: (JSVal -> IO ()) -> OnBlocked -> IO (Callback (JSVal -> IO ()))`: Synchronous callbacks that don't return a value. These take an additional `data OnBlocked = ThrowWouldBlock | ContinueAsync` argument for use in the case that the thread becomes blocked on e.g. an `MVar` transaction.
- `syncCallback' :: (JSVal -> IO JSVal) -> IO (Callback (JSVal -> IO ()))`: Synchronous callbacks that return a value. Because of the return value, there is no possibility of continuing asynchronously, so no `OnBlocked` argument is taken.
- `asyncCallback :: (JSVal -> IO ()) -> IO (Callback (JSVal -> IO ()))`: Asynchronous callbacks that immediately start in a new thread. Cannot return a value.

There is no checking that the passed arguments match the callback, so the following example compiles and correctly prints 10, despite the argument being passed as an `Int` to a `Callback` that accepts a `JSVal`:

```haskell
foreign import javascript "((f,x) => { return f(x); })"
  apply_int :: Callback (JSVal -> IO JSVal) -> Int -> IO Int

main :: IO ()
main = do
  add3 <- syncCallback1' (return . (+3))
  print =<< apply_int add3 7
  releaseCallback add3
```

14.1.3 Callbacks as Foreign Exports

JavaScript callbacks allow for a sort of FFI exports via FFI imports. To do this, a global JavaScript variable is set, and that global variable can then be called from use cases that access plain JavaScript functions - such as interactive HTML elements. This would look like:

```haskell
foreign import javascript "((f) => { globalF = f })"
  setF :: Callback (JSVal -> IO ()) -> IO ()

main :: IO ()
main = do
  log <- syncCallback1 ThrowWouldBlock (print . fromJSString)
  setF log
  -- don't releaseCallback log
```

```html
<button onClick="globalF('Button pressed!")>Example</button>
```

We have to make sure not to use `releaseCallback` on any functions that are to be available in HTML, because we want these functions to be in memory indefinitely.

14.2 Writing Replacement Implementations for Libraries with C FFI Functions

Many libraries make use of C FFI functions to accomplish low-level or performance sensitive operations - known as `cbits` and often kept in a folder with this name. For such a library to support the JavaScript backend, the `cbits` must have replacement implementations.

In principle, it is possible for the JavaScript backend to automatically compile `cbits` using Emscripten, but this requires wrappers to convert data between the JS backend's RTS data format, and the format expected by Emscripten-compiled functions. Since C functions are often used where performance is more critical, there's potential for the data conversions to negate this purpose.

Instead, it is more effective for a library to provide an alternate implementation for functions using the C FFI - either by providing direct one-to-one replacement JavaScript functions, or by using C preprocessor directives to replace C FFI imports with some combination of JS FFI imports and pure-Haskell implementation.

14.2.1 Direct Implementation of C FFI Imports in JavaScript as `jsbits`

When the JavaScript backend generates code for a C FFI import, it will call the function named in the import string, prepended by `h$` - so the imported C function `open` will look for the JavaScript function `h$open`. No verification is done to ensure that these functions are actually implemented in the linked JavaScript files, so there can be runtime errors when a missing JavaScript function is called.

Based on this, implementing a C function in JavaScript is a matter of providing a function of the correct shape (based on the C FFI import type signature) in any of the linked JavaScript sources. External JavaScript sources are linked by either providing them as an argument to GHC, or listing them in the `js-sources` field of the cabal file - in which case it would usually be inside a predicate to detect the `javascript` architecture, such as:

```
library

  if arch(javascript)
    js-sources:
      jsbits/example.js
```

Note that `js-sources` requires Cabal 3.10 to be used with library targets, and Cabal 3.12 to be used with executable targets.

The shape required of the JavaScript function will depend on the particular C types used:

- primitives, such as `CInt` will map directly to a single JavaScript argument using JavaScript primitives. In the case of `CInt`, this will be a JavaScript number. Note that in the case of return values, a JavaScript number will usually need to be rounded or cast back to an integral value in cases where mathematical operations are used
- pointer values, including `CString`, are passed as an unboxed `(ptr, offset)` pair. For arguments, being unboxed will mean these are passed as two top-level arguments to the function. For return values, unboxed values should be returned from JavaScript functions by using a special C preprocessor macro, `RETURN_UBX_TUP2(ptr, offset)`
- `CString`, in addition to the above pointer handling, will need to be decoded and encoded to convert them between character arrays and JavaScript strings.

- other RTS primitive types are discussed previously in *JavaScript FFI Types* (page 711).

As an example, let's consider the implementation of `getcwd`:

```
-- unix:System.Posix.Directory

foreign import ccall unsafe "getcwd" c_getcwd :: Ptr CChar -> CSize -> IO (Ptr CChar)
```

```
// libraries/base/jsbits/base.js

//#OPTIONS: CPP

function h$getcwd(buf, off, buf_size) {
  try {
    var cwd = h$encodeUtf8(process.cwd());
    if (buf_size < cwd.len && buf_size !== 0) {
      h$setErrno("ERANGE");
      RETURN_UBX_TUP2(null, 0);
    } else if (buf !== null) {
      h$copyMutableByteArray(cwd, 0, buf, off, cwd.len);
      RETURN_UBX_TUP2(buf, off);
    } else if (buf_size === 0) {
      RETURN_UBX_TUP2(cwd, 0);
    } else {
      var out = h$newByteArray(buf_size);
      h$copyMutableByteArray(cwd, 0, out, off, cwd.len);
    }
  } catch (e) {
    h$setErrno(e);
    RETURN_UBX_TUP2(null, 0);
  }
}
```

Here, the C function `getcwd` maps to the JavaScript function `h$getcwd`, which exists in a `.js` file within `base`'s `jsbits` subdirectory. `h$getcwd` expects a `CString` (passed as the equivalent `Ptr CChar`) and a `CSize` argument. This results in three arguments to the JavaScript function - two for the string's pointer and offset, and one for the size, which will be passed as a JavaScript number.

Next, the JavaScript `h$getcwd` function demonstrates several details:

- In the try clause, the `cwd` value is first accessed using a NodeJS-provided method. This value is immediately encoded using `h$encodeUtf8`, which is provided by the JavaScript backend. This function will only return the pointer for the encoded value, and the offset will always be 0
- Next, we select one of several cases - based on the specification of the C function that we're trying to immitate
- In the first case where the given buffer size is too small, but not zero, the function must set the `ERANGE` error code, which we do here with `h$setErrno`, and return `null`. As we saw in the function arguments, pointers are passed as a `(ptr, offset)` pair - meaning `null` is represented by returning the unboxed pair `(null, 0)`
- In the second case where there is enough space in `buf` to successfully copy the bytes, we do so using `h$copyMutableByteArray` - a function supplied by GHC's JavaScript RTS
- In the third case where `buf_size` is 0, this indicates in the C function's specification that we can allocate a new buffer of the appropriate size to return. We already have this in the form of the previously encoded `cwd`, so we can just return it, along with the 0 offset

- In the last case where `buf` is null, and `buf_size` is large enough, we allocate a new buffer, this time with `buf_size` bytes of space using `h$newByteArray`, and we again perform a mutable copy
- To use C preprocessor macros in linked JavaScript files, the file must open with the `// #OPTIONS: CPP` line, as is shown towards the start of this snippet
- If an error occurs, the catch clause will pass it to `h$setErrno` and return the `(null, 0)` pointer and offset pair - which is a behaviour expected by the C function in the error case.

14.2.2 Writing JavaScript Functions to be NodeJS and Browser Aware

In the above example of implementing `getcwd`, the function we use in the JavaScript implementation is from NodeJS, and the behaviour doesn't make sense to implement in a browser. Therefore, the actual implementation will include a C preprocessor condition to check if we're compiling for the browser, in which case `h$unsupported(-1)` will be called. There can be multiple non-browser JavaScript runtimes, so we'll also have to check at runtime to make sure that NodeJS is in use.

```
function h$getcwd(buf, off, buf_size) {
#ifndef GHCJS_BROWSER
  if (h$isNode()) {
    try {
      var cwd = h$encodeUtf8(process.cwd());
      if (buf_size < cwd.len && buf_size !== 0) {
        h$setErrno("ERANGE");
        return (null, 0);
      } else if (buf !== null) {
        h$copyMutableByteArray(cwd, 0, buf, off, cwd.len);
        RETURN_UBX_TUP2(buf, off);
      } else if (buf_size === 0) {
        RETURN_UBX_TUP2(cwd, 0);
      } else {
        var out = h$newByteArray(buf_size);
        h$copyMutableByteArray(cwd, 0, out, off, cwd.len);
      }
    } catch (e) {
      h$setErrno(e);
      RETURN_UBX_TUP2(null, 0);
    }
  } else
#endif
    h$unsupported();
    RETURN_UBX_TUP2(null, 0);
}
```

14.2.3 Replacing C FFI Imports with Pure Haskell and JavaScript

Instead of providing a direct JavaScript implementation for each C FFI import, we can instead use the C preprocessor to conditionally remove these C imports (and possibly use sites as well). Then, some combination of JavaScript FFI imports and Haskell implementation can be added instead. As in the direct implementation section, any linked JavaScript files should usually be in a `if arch(javascript)` condition in the cabal file.

As an example of a mixed Haskell and JavaScript implementation replacing a C implementation, consider `base:GHC.Clock`:

```
#if defined(javascript_HOST_ARCH)
getMonotonicTimeNSec :: IO Word64
getMonotonicTimeNSec = do
  w <- getMonotonicTimeMSec
  return (floor w * 1000000)

foreign import javascript unsafe "performance.now"
  getMonotonicTimeMSec :: IO Double

#else
foreign import ccall unsafe "getMonotonicNSec"
  getMonotonicTimeNSec :: IO Word64
#endif
```

Here, the `getMonotonicTimeNSec` C FFI import is replaced by the JavaScript FFI import `getMonotonicTimeMSec`, which imports the standard JavaScript function `performance.now`. However, because this JavaScript implementation returns the time as a `Double` of floating point milliseconds, it must be wrapped by a Haskell function to extract the integral value that's expected.

In this case, the choice of using a mixed Haskell and JavaScript replacement implementation was caused by the limitation of clocks being system calls. In a lot of cases, C functions are used for similar system-level functionality. In such cases, it's recommended to import the required system functions from standard JavaScript libraries (or from the runtime, as was required for `getcwd`), and use Haskell wrapper functions to convert the imported functions to the appropriate format.

In other cases, C functions are used for performance. For these cases, pure-Haskell implementations are the preferred first step for compatibility with the JavaScript backend since it would be more future-proof against changes to the RTS data format. Depending on the use case, compiler-optimised JS code might be hard to complete with using hand-written JavaScript. Generally, the most likely performance gains from hand-written JavaScript come from functions with data that stays as JavaScript primitive types for a long time, especially strings. For this, `JSVal` allows values to be passed between `Haskell` and `JavaScript` without a marshalling penalty.

14.3 Linking with C sources

GHC supports compiling C sources into JavaScript (using Emscripten) and linking them with the rest of the JavaScript code (generated from Haskell codes and from the RTS).

C functions compiled with Emscripten get a "_" prepended to their name in JavaScript. For example, C "malloc" becomes "_malloc" in JavaScript.

14.3.1 EMCC pragmas

By default the EMCC linker drops code considered dead and it has no way to know which code is alive due to some call from Haskell or from a JavaScript wrapper. As such, you must explicitly add some pragmas at the top of one of your *.js* files to indicate which functions are alive:

` //#OPTIONS:EMCC:EXPORTED_RUNTIME_METHODS=foo,bar `

Enable methods *foo* and *bar* from the Emscripten runtime system. This is used for methods such as *ccall*, *cwrap*, *addFunction*, *removeFunction*… that are described in Emscripten documentation.

` //#OPTIONS:EMCC:EXPORTED_FUNCTIONS=_foo,_bar `

Enable C functions *foo* and *bar* to be exported respectively as *_foo* and *_bar* (_ prepended). This is used for C library functions (e.g. *_malloc*, *_free*, etc.) and for the C code compiled with your project (e.g. *_sqlite3_open* and others for the *sqlite* C library).

You can use both pragmas as many times as you want. Ultimately all the entries end up in sets of functions passed to the Emscripten linker via *-sEXPORTED_RUNTIME_METHODS* and *-sEXPORTED_FUNCTIONS* (which can only be passed once; the latter argument overrides the former ones).

` //#OPTIONS:EMCC:EXTRA=-foo,-bar `

This pragma allows additional options to be passed to Emscripten if need be. We already pass:

- *-sSINGLE_FILE=1*: required to create a single *.js* file as artefact (otherwise *.wasm* files corresponding to C codes need to be present in the current working directory when invoking the resulting *.js* file).
- *-sALLOW_TABLE_GROWTH*: required to support *addFunction*
- *-sEXPORTED_RUNTIME_METHODS* and *-sEXPORTED_FUNCTIONS*: see above.

Be careful because some extra arguments may break the build in unsuspected ways.

14.3.2 Wrappers

The JavaScript backend doesn't generate wrappers for foreign imports to call directly into the compiled C code. I.e. given the following foreign import:

`haskell foreign import ccall "foo" foo :: ... `

The JavaScript backend will replace calls to *foo* with calls to the JavaScript function *h$foo*. It's still up to the programmer to call *_foo* or not from *h$foo* on a case by case basis. If *h$foo* calls the generated from C function *_foo*, then we say that *h$foo* is a wrapper function. These wrapper functions are used to marshal arguments and returned values between the JS heap and the Emscripten heap.

On one hand, GHC's JavaScript backend creates a different array of bytes per allocation (in order to make use of the garbage collector of the JavaScript engine). On the other hand, Emscripten's C heap consists in a single array of bytes. To call C functions converted to JavaScript that have pointer arguments, wrapper functions have to:

1. allocate some buffer in the Emscripten heap (using *_malloc*) to get a valid Emscripten pointer
2. copy the bytes from the JS object to the buffer in the Emscripten heap
3. use the Emscripten pointer to make the call to the C function
4. optionally copy back the bytes from the Emscripten heap if the call may have changed the contents of the buffer
5. free the Emscripten buffer (with *_free*)

GHC's JavaScript rts provides helper functions for this in *rts/js/mem.js*. See *h$copyFromHeap*, *h$copyToHeap*, *h$initHeapBuffer*, etc.

14.3.3 Callbacks

Some C functions take function pointers as arguments (e.g. callbacks). This is supported by the JavaScript backend but requires some work from the wrapper functions.

1. On the Haskell side it is possible to create a pointer to an Haskell function (a *FunPtr*) by using a "wrapper" foreign import. See the documentation of *base:Foreign.Ptr.FunPtr*.
2. This *FunPtr* can be passed to a JavaScript wrapper function. However it's implemented as a *StablePtr* and needs to be converted into a function pointer that Emscripten understands. This can be done with *h$registerFunPtrOnHeap*.
3. When a callback is no longer needed, it can be freed with *h$unregisterFunPtrFromHeap*.

Note that in some circumstances you may not want to register an Haskell function directly as a callback. It is perfectly possible to register/free regular JavaScript function as Emscripten functions using *Module.addFunction* and *Module.removeFunction*. That's what the helper functions mentioned above do.

CHAPTER

FIFTEEN

USING THE GHC WEBASSEMBLY BACKEND

15.1 What does the WebAssembly "backend" mean

In order to compile Haskell to wasm, you need a custom GHC build that targets wasm. There isn't a GHC option which allows you to use a stock GHC installed via `ghcup` or `stack` to generate wasm. That's because GHC is still a single-target compiler, so each GHC build is only capable of compiling code that runs on a single architecture and operating system.

So, the term GHC wasm backend isn't the same sense as the unregisterised/LLVM/NCG backends. It merely describes GHC's support as a cross compiler that targets wasm, and more specifically, `wasm32-wasi`.

The generated wasm module makes use of a few post-MVP extensions that are supported by default in latest releases of Chrome/Firefox/Safari/wasmtime. The wasm module uses WASI as the system call layer, so it's supported by any wasm engine that implements WASI (including browsers, which can provide the WASI layer via JavaScript).

15.2 Setting up the GHC wasm backend

The wasm backend is still a tech preview and not included in the official bindists yet. If you are using x86_64-linux, you can follow the "getting started" subsections in ghc-wasm-meta to quickly set up the GHC wasm backend using the nightly artifacts.

It's also possible to build the GHC wasm backend manually, if your host system is one of {x86_64,aarch64}-{linux,darwin}. Refer to the `ghc-wasm-meta` readme for detailed instructions.

15.3 Using the GHC wasm backend to compile & link code

Once the GHC wasm backend is set up, you can use it to compile and link code. The compiler executables follow the cross compiler linking convention, so you need to call `wasm32-wasi-ghc`, `wasm32-wasi-ghc-pkg` and `wasm32-wasi-hsc2hs` instead of `ghc`, `ghc-pkg` and `hsc2hs`.

You can also use the `--with-compiler=`, `--with-hc-pkg=` and `--with-hsc2hs` flags of `cabal` to build cabal projects. The `wasm32-wasi-cabal` wrapper script set up by the `ghc-wasm-meta` installer does this automatically for you, but using flags manually also works with stock `cabal` installations. When `cabal` builds an executable component, that executable will be built as a wasm module, and you can use `cabal list-bin exe:foo` to find the wasm module's location in the build directory.

15.4 Running the GHC wasm backend's output

Once you have a wasm module, you can run it with a dedicated wasm engine like `wasmtime`, or inside the browsers.

To run it with `wasmtime`, you can simply do:

```
$ wasmtime run foo.wasm
```

Just like native executables, you can pass command line arguments, and also RTS options, as long as it's built with `-rtsopts`:

```
$ wasmtime run foo.wasm --bar +RTS --nonmoving-gc -RTS
```

You can also mount some host directory into it:

```
$ wasmtime run --mapdir /::$PWD foo.wasm
```

As long as the filesystem capability is provided, in addition to filesystem I/O in Haskell code, you can use the RTS eventlog and profiling functionality, then inspect the report files:

```
$ wasmtime run --mapdir /::$PWD foo.wasm +RTS -hc -l -RTS
```

15.5 JavaScript FFI in the wasm backend

The GHC wasm backend supports the JavaScript FFI feature. For Haskell projects that are meant to be run in JavaScript environments like browsers or nodejs, the JavaScript FFI enables:

- Calling JavaScript from Haskell via foreign imports and vice versa via foreign exports.
- Representing JavaScript values as first-class garbage collected Haskell values on the Haskell heap.
- Blocking for asynchronous JavaScript computation in a single Haskell thread without blocking the entire runtime.
- Not paying for JavaScript when not using it, the same toolchain still generates self-contained `wasm32-wasi` modules by default.

The JavaScript FFI as implemented in the GHC wasm backend is pioneered by the asterius project and heavily inspired by GHCJS, the predecessor of the GHC JavaScript backend. Despite some similarities, it still differs from the GHC JavaScript backend's implementation significantly. The rest of this guide is a canonical reference for the GHC wasm backend's JavaScript FFI, which we'll now abbreviate as JSFFI.

15.5.1 Marshalable types and `JSVal`

JSFFI supports all boxed marshalable foreign types in C FFI:

- `Bool`
- `Char`
- `Int` / `Word`
- `Int8` / `Int16` / `Int32` / `Int64`
- `Word8` / `Word16` / `Word32` / `Word64`
- `Ptr` / `FunPtr` / `StablePtr`
- `Float` / `Double`

The above types and their `newtypes` can be used as argument/result types in JSFFI. Some caveats to keep in mind:

- `Bool` is marshaled to `0 / 1` instead of `false / true` in JavaScript. This is affected by implementation details of JSFFI, which is layered on top of C FFI and shares some characteristics of C FFI. It should be fine in most cases, since implicit conversion to `boolean` happens when it's used as a boolean. It's also fine to pass a JavaScript `boolean` into Haskell, since it'll be implicitly converted to a number first.
- Likewise, `Char` is marshaled to 32-bit integer that represents its Unicode code point. Do not pass a single character JavaScript `string` as `Char`, since implicit conversion to number results in `NaN`! If you absolutely need to use `Char` as a JSFFI argument/result type, you're in charge of handling `Char`s as code points. Most likely you only need to marshal between Haskell `String` or `Text` and JavaScript `string`s, for which there already exist conversion functions.
- 64-bit integer types are marshaled to JavaScript `bigint`s. In JavaScript, mixing `bigint` and regular numbers in arithmetic results in type errors, so keep this in mind. As for `Int` / `Word`, they are 32-bit since the GHC wasm backend is based on `wasm32` .
- JSFFI doesn't support unboxed foreign types like `Int#`, `ByteArray#`, etc, even when `UnliftedFFITypes` is enabled.

In addition to the above types, JSFFI supports the `JSVal` type and its `newtypes` as argument/result types. `JSVal` is defined in `GHC.Wasm.Prim` in `ghc-experimental`, which represents an opaque reference to a JavaScript value.

`JSVal`s are first-class Haskell values on the Haskell heap. You can get them via foreign import results or foreign export arguments, store them in Haskell data structures and pass them between Haskell/JavaScript. They are garbage-collected by the GHC RTS:

- There can be multiple `JSVal`s that point to the same JavaScript value. As long as there's at least one `JSVal` still alive on the Haskell heap, that JavaScript value will still be alive on the JavaScript heap.
- If there's no longer any live `JSVal` that points to the JavaScript value, then after Haskell garbage collection, the runtime no longer retain any reference to it, allowing the JavaScript runtime to eventually garbage collect it as well.

In addition to garbage collection, `GHC.Wasm.Prim` also exports `freeJSVal :: JSVal -> IO ()`, allowing the user to drop the JavaScript reference from the runtime eagerly. You're encouraged to make use of `freeJSVal` when you're sure about a `JSVal`'s lifetime, especially for the temporary `JSVal`s. This will help reducing the memory footprint at runtime.

Note that `freeJSVal` is not idempotent and it's only safe to call it exactly once or not at all. Once it's called, any subsequent usage of that `JSVal` results in a runtime panic.

15.5.2 Foreign imports

One can embed a JavaScript code snippet in a foreign import declaration and call that piece of JavaScript code by calling the foreign import function:

```
import GHC.Wasm.Prim

foreign import javascript unsafe "console.log($1)"
  js_print :: JSString -> IO ()

foreign import javascript unsafe "typeof $1 === 'object'"
  js_is_obj :: JSVal -> Bool

foreign import javascript unsafe "let acc = 1; for (let i = 1; i <= $1; ++i) acc *= i;
↪ return acc;"
  js_fac :: Word -> Word
```

A JSFFI import code snippet can be either a single JavaScript expression or a series of JavaScript statements as function body, in which case you can use `return` to return the import result value. The import code snippet has access to:

- The import argument values, bound to arguments `$1`, `$2`, etc.
- The `__export` binding, which contain all wasm module exports. For instance, you could use `__exports.memory` to access the `WebAssembly.Memory` object and use it to copy blobs between the Haskell/JavaScript side. The `memory` export exists by default.
- The full Web API that exists in the JavaScript global scope.

There are two kinds of JSFFI imports: synchronous/asynchronous imports. `unsafe` indicates synchronous imports, which has the following caveats:

- The calling thread as well as the entire runtime blocks on waiting for the import result.
- If the JavaScript code throws, the runtime crashes with the same error. A JavaScript exception cannot be handled as a Haskell exception here, so you need to use a JavaScript `catch` explicitly shall the need arise.
- Like `unsafe` C imports, re-entrance is not supported, the imported foreign code must not call into Haskell again. Doing so would result in a runtime panic.

When a JSFFI import is marked as `safe` / `interruptible` or lacks safety annotation, then it's treated as an asynchronous import. The asynchronous JSFFI imports combine the Haskell concurrency model and the JavaScript event loop, allowing Haskell code to work with async JavaScript computation without blocking the entire runtime.

```
import Control.Exception

foreign import javascript safe "new Promise(res => setTimeout(res, $1))"
  js_sleep :: Int -> IO ()

sleep :: Int -> IO ()
sleep t = evaluate =<< js_sleep t

foreign import javascript safe "const r = await fetch($1); return r.text();"
  js_fetch :: JSString -> IO JSString
```

Asynchronous import code is wrapped in async JavaScript functions, therefore `await` is also supported. Async JavaScript functions always return `Promises`, and you can also explicitly create and return a `Promise` that resolves to the final result of the async computation.

When an asynchronous JSFFI import is called, the Haskell function returns immediately once the async JavaScript function returns a `Promise`. The value returned by the Haskell function is a thunk. When the thunk is evaluated later, the evaluating thread is suspended by the runtime, and resumed when the `Promise` actually resolves or rejects.

Compared to synchronous JSFFI imports, asynchronous JSFFI imports have the following notable pros/cons:

- Waiting for the result only blocks a single Haskell thread, other threads can still make progress and garbage collection may still happen.
- If the `Promise` rejects, Haskell code can catch JavaScript errors as `JSExceptions`.
- Re-entrance is supported. The JavaScript code may call into Haskell again and vice versa.
- Of course, it has higher overhead than synchronous JSFFI imports.

Using thunks to encapsulate `Promise` result allows cheaper concurrency without even needing to fork Haskell threads just for waiting for a bunch of async calls to return. Just like lazy I/O, the convenience comes with caveat, you need to take some care to force the result thunk before closing the underlying resource. And even if the result type is `()`, it's still a thunk that needs to be explicitly forced to ensure the `Promise` has actually resolved, so you likely need to write a worker/wrapper function pair for cases like `sleep`.

There's also a special kind of JSFFI import that allow converting a callable `JSVal` to a Haskell function:

```
type Logger = JSString -> IO ()

type JSFunction = JSVal

foreign import javascript unsafe "s => console.log(s)"
  js_logger :: JSFunction

foreign import javascript unsafe "dynamic"
  js_logger_to_hs :: JSFunction -> Logger
```

Much like `foreign import ccall "dynamic"` which wraps a C function pointer as a Haskell function, `foreign import javascript "dynamic"` wraps a `JSVal` that represent a JavaScript function as a Haskell function. The returned Haskell function retains the reference to that `JSVal`, and the `unsafe` / `safe` annotation indicates whether that JavaScript function is synchronous or asynchronous.

Of course, without `foreign import javascript "dynamic"`, one could still easily implement similar functionality:

```
foreign import javascript unsafe "$1($2)"
  js_logger_to_hs :: JSFunction -> JSString -> IO ()
```

And that's how it's implemented under the hood. It's handled as a JSFFI import with an auto-generated code snippet that calls the first argument, passing the rest of arguments.

15.5.3 Foreign exports

One can use `foreign export javascript` to export a top-level Haskell binding as a wasm module export which can be called in JavaScript:

```
foreign export javascript "my_fib"
  fib :: Word -> Word
```

Give `fib :: Word -> Word`, the above declaration exports `fib` as `my_fib`. It is a wasm module export function without any JavaScript wrapper, and as long as the wasm instance is properly initialized, you can call `await instance.exports.my_fib(10)` to invoke the exported Haskell function and get the result.

Unlike JSFFI imports which have synchronous/asynchronous flavors, JSFFI exports are always asynchronous. Calling them always return a `Promise` in JavaScript that needs to be `await`ed for the real result. If the Haskell function throws, the `Promise` is rejected with a `WebAssembly.RuntimeError`, and the `message` field contains a JavaScript string of the Haskell exception.

Above is the static flavor of JSFFI exports. It's also possible to export a dynamically created Haskell function closure as a JavaScript function and obtain its `JSVal`:

```
type BinOp a = a -> a -> a

foreign import javascript "wrapper"
  js_func_from_hs :: BinOp Int -> IO JSVal
```

This is also much like `foreign import ccall "wrapper"`, which wraps a Haskell function closure as a C function pointer. Note that `unsafe` / `safe` annotation is ignored here, since the `JSVal` that represent the exported function is always returned synchronously, but it is always an asynchronous JavaScript function, just like static JSFFI exports.

The `JSVal` callbacks created by dynamic JSFFI exports can be passed to the rest of JavaScript world to be invoked later. But wait, didn't we say earlier that `JSVal`s are garbage collected? Isn't a use-after-free trap waiting ahead of the road, when the `JSVal` is collected in Haskell but the JavaScript callback is invoked later?

So, normal `JSVal`s created by JSFFI import results or JSFFI export arguments only manage a single kind of resource: the JavaScript value it refers to. But `JSVal`s created by dynamic JSFFI exports manage two kinds of resources: the JavaScript callback it refers to, as well as a stable pointer that retains the Haskell function closure. If this `JSVal` is garbage collected, the Haskell runtime no longer retains the JavaScript callback, but the JavaScript side may still hold that callback and intends to call it later, so the Haskell function closure is still retained by default.

Still, the runtime can gradually drop these retainers by using `FinalizerRegistry` to invoke the finalizers to free the underlying stable pointers once the JavaScript callbacks are recycled.

One last corner case is cyclic reference between the two heaps: if a JavaScript callback is retained only by `JSVal` and that `JSVal` is retained only by a Haskell function closure that gets exported, this creates a cyclic reference that can't be automatically recycled. This is a fundamental limit of the GHC wasm backend today since the Haskell heap lives in the linear memory, distinct from the host JavaScript heap, and coordination between two heaps is always a non-trivial challenge. However, one can still use `freeJSVal` to break the cycle. When `freeJSVal` is applied to a `JSVal` that represents a JavaScript callback created by a dynamic JSFFI export, both kinds of resources are freed at once: the JavaScript callback, as well as the Haskell function closure.

15.5.4 Detect whether JSFFI is being used

If you're writing a Haskell library, you may want to detect whether the final linked module involves JSFFI logic, or is it still a self-contained `wasm32-wasi` module. One obvious reason is `wasi` implementations in JavaScript environments are often incomplete and lack certain features (e.g. the `poll_oneoff` syscall), so it makes sense to dispatch code.

`GHC.Wasm.Prim` exports `isJSFFIUsed :: Bool` that can be used for this purpose. As long as there's a single JSFFI import/export or anything involving `JSVal` linked into the final wasm module, it will be `True`. It is `False` if and only if the user code has absolutely no transitive dependency on anything related to JSFFI, in which case the linked wasm module will be a self-contained `wasm32-wasi` module. If something compiles fine with the GHC wasm backend before JSFFI feature is merged, but `isJSFFIUsed` is still `True`, then it's definitely a bug.

15.5.5 Interaction with async exception

When a thread is blocked waiting for an async JSFFI import call to return, it can be interrupted by a Haskell async exception, with some caveats:

- The async exception would not magically cancel the `Promise`. In general, JavaScript `Promises` aren't cancelable anyway. There do exist third-party `Promise` libraries to provide a "cancel" interface, which only guarantees all `.then()` continuations registered on that `Promise` will no longer be invoked. For simplicity of implementation, we aren't using those for the time being.
- Normally, `throwTo` would block until the async exception has been delivered. In the case of JSFFI, `throwTo` would always return successfully immediately, while the target thread is still left in a suspended state. The target thread will only be waken up when the `Promise` actually resolves or rejects, though the `Promise` result will be discarded at that point.

The current way async exceptions are handled in JSFFI is subject to change though. Ideally, once the exception is delivered, the target thread can be waken up immediately and continue execution, and the pending `Promise` will drop reference to that thread and no longer invoke any continuations.

15.5.6 Interaction with C FFI

User code can take advantage of JSFFI and C FFI together, and make use of third party C/C++ code as long as they work on `wasm32-wasi`. However, there is an important limitation to keep in mind when it comes to the interaction between JSFFI and C FFI:

A Haskell thread cannot force an async JSFFI import thunk when it represents a Haskell function exported via C FFI. Doing so would throw `WouldBlockException`.

For example, suppose we're using the Haskell bindings of a certain C library, and some of the C functions expect callers to pass a C function pointer as the callback argument. Yes, we can use `foreign import ccall "wrapper"` to wrap a Haskell function closure and pass it to that C function. The wrapped Haskell function can even call sync JSFFI imports, but it cannot call an async JSFFI import and block on the result.

The other direction is fine. Regardless of whether a C import is `unsafe` or `safe`, it can be called in a Haskell thread that represents a Haskell function exported to JavaScript. Do keep in mind that we're using the single-threaded runtime at the moment, so other than supporting re-entrancy, `safe` C calls don't offer extra advantage than `unsafe`.

As mentioned before, JSFFI is layered on top of C FFI under the hood, and they share the same C symbol namespace. In most cases, the JSFFI related symbols are auto-generated so they can't collide, but for each `foreign export javascript "my_func"`, there will be a `my_func` externally visible C symbol, so you need to take a bit of care not to duplicate symbol with the C side.

15.5.7 The JavaScript API

When linking a wasm module that makes use of JSFFI, correct link-time arguments must be passed to GHC and this needs to be adjusted on a per-project basis:

```
ghc -no-hs-main -optl-mexec-model=reactor -optl-Wl,--export=my_func
```

Why?

Consider the case `ghc hello.hs` where `hello.hs` is just a good old `main = putStrLn "hello world"`. By default, ghc exports `main` as a C function, compiles and links a little C stub file that contain the actual `main` that initializes the runtime and call the Haskell `main`, and `clang` would link everything as a `wasm32-wasi` command module.

Conceptually, a `wasm32-wasi` command module is like an executable, with a single entry point and meant to be invoked only once and then exits. But certain link-time arguments can tell `clang` to target a `wasm32-wasi` reactor module instead. A reactor module is a bit like a shared library: it has internal functions as well as user-defined entry points, and once it's initialized, the entry points can be called as many times as the user wants.

Given the nature of JSFFI, if a project uses JSFFI, then it surely is meant to target a `wasm32-wasi` reactor module. And there's no sensible default entry point, not even `main`, you need to explicitly pass the export names you need via link-time arguments, otherwise those exports will be absent from the resulting wasm module due to linker dead code elimination.

Now, suppose a wasm module has already been linked and it makes use of JSFFI. This wasm module will contain `ghc_wasm_jsffi` custom sections, and the section payloads include information like JSFFI import code snippets and function arities. The next step is calling a small post-linker script, which will parse the wasm module and emit a JavaScript module. The post-linker is written in nodejs, though the resulting JavaScript module has nothing nodejs specific and work in browsers as well.

```
$ $(wasm32-wasi-ghc --print-libdir)/post-link.mjs -i hello.wasm -o hello.js
```

The generated JavaScript module contains a default export which is a function. The function takes an `__exports` argument and generates the `ghc_wasm_jsffi` wasm imports. There is knot-tying going on here: only after a wasm module is instantiated, you can access its exports, but the `ghc_wasm_jsffi` imports required for instantiation need access to the exports! JavaScript is not a lazy language, but we can achieve knot-tying less elegantly by using mutation:

```
let __exports = {};

const { instance } = await WebAssembly.instantiateStreaming(
  fetch(wasm_url),
  {
    ghc_wasm_jsffi: (await import(js_url)).default(__exports),
    wasi_snapshot_preview1: ...
  }
);
```

(continues on next page)

(continued from previous page)

```
Object.assign(__exports, wasm_instance.exports);
```

This way, the `ghc_wasm_jsffi` imports will have access to all exports of the wasm instance.

After the wasm instance is created, initialization needs to be done:

```
wasi.initialize(wasm_instance);
```

The `wasm32-wasi` reactor module ABI defines the `_initialize` export function, which is auto generated at link time and it must be called exactly once before any other wasm exports can be called. The correct way to call it depends on the wasi implementation provider in JavaScript.

Finally, in JavaScript, you can use `await __exports.my_func()` to call your exported `my_func` function and get its result, pass arguments, do error handling, etc etc.

CHAPTER

SIXTEEN

KNOWN BUGS AND INFELICITIES

16.1 Haskell standards vs. Glasgow Haskell: language non-compliance

This section lists Glasgow Haskell infelicities in its implementation of Haskell 98 and Haskell 2010. See also the "when things go wrong" section (*What to do when something goes wrong* (page 689)) for information about crashes, space leaks, and other undesirable phenomena.

The limitations here are listed in Haskell Report order (roughly).

16.1.1 Divergence from Haskell 98 and Haskell 2010

GHC aims to be able to behave (mostly) like a Haskell 98 or Haskell 2010 compiler, if you tell it to try to behave like that with the *Haskell98* (page 272) and *Haskell2010* (page 272) flags. The known deviations from the standards are described below. Unless otherwise stated, the deviation applies in both Haskell 98 and Haskell 2010 mode.

16.1.1.1 Lexical syntax

- Certain lexical rules regarding qualified identifiers are slightly different in GHC compared to the Haskell report. When you have ⟨module⟩.⟨reservedop⟩, such as `M.\`, GHC will interpret it as a single qualified operator rather than the two lexemes `M` and `.\`.
- `forall` is always a reserved keyword. This is contrary to the Haskell Report, which allows variables and type variables to be named `forall`. Note that this does not imply that GHC always enables the *ExplicitForAll* (page 506) extension. Even without this extension enabled, reserving `forall` as a keyword has significance. For instance, GHC will not parse the type signature `foo :: forall x`.
- The `(!)` operator, when written in prefix form (preceded by whitespace and not followed by whitespace, as in `f !x = ...`), is interpreted as a bang pattern, contrary to the Haskell Report, which prescribes to treat `!` as an operator regardless of surrounding whitespace. Note that this does not imply that GHC always enables *BangPatterns* (page 541). Without the extension, GHC will issue a parse error on `f !x`, asking to enable the extension.
- Irrefutable patterns must be written in prefix form:

```
f ~a ~b = ...    -- accepted by both GHC and the Haskell Report
f ~ a ~ b = ...  -- accepted by the Haskell Report but not GHC
```

When written in non-prefix form, `(~)` is treated by GHC as a regular infix operator.

See GHC Proposal #229 for the precise rules.

- Strictness annotations in data declarations must be written in prefix form:

```
data T = MkT !Int   -- accepted by both GHC and the Haskell Report
data T = MkT ! Int  -- accepted by the Haskell Report but not GHC
```

See GHC Proposal #229 for the precise rules.

- As-patterns must not be surrounded by whitespace on either side:

```
f p@(x, y, z) = ...    -- accepted by both GHC and the Haskell Report

-- accepted by the Haskell Report but not GHC:
f p @ (x, y, z) = ...
f p @(x, y, z) = ...
f p@ (x, y, z) = ...
```

When surrounded by whitespace on both sides, `(@)` is treated by GHC as a regular infix operator.

When preceded but not followed by whitespace, `(@)` is treated as a visible type application.

See GHC Proposal #229 for the precise rules.

- Haskell Report allows any Unicode Decimal Number in decimal literals. However, GHC accepts only ASCII numbers:

```
ascDigit    →   0 | 1 | … | 9
decimal     →   ascDigit {ascDigit}
```

- GHC is more lenient in which characters are allowed in the identifiers. Unicode Other Letters are considered to be small letters, therefore variable identifiers can begin with them. Digit class contains all Unicode numbers instead of just Decimal Numbers. Modifier Letters and Non-Spacing Marks can appear in the tail of the identifiers.:

```
uniSmall    →   any Unicode Lowercase Letter or Other Letter
uniDigit    →   any Unicode Decimal Number, Letter Number or Other Number

uniIdchar   →   any Unicode Modifier Letter or Non-Spacing Mark
idchar      →   small | large | digit | uniIdchar | '

varid       →   small {idchar} ⟨reservedid⟩
conid       →   large {idchar}
```

- GHC allows redundant parantheses around the function name in the *funlhs* part of declarations. That is GHC will succeed in parsing a declaration like *((f)) x = <rhs>* for any number of parantheses around *f*.

16.1.1.2 Context-free syntax

- In Haskell 98 mode (but not in Haskell 2010 mode), GHC is a little less strict about the layout rule when used in `do` expressions. Specifically, the restriction that "a nested context must be indented further to the right than the enclosing context" is relaxed to allow the nested context to be at the same level as the enclosing context, if the enclosing context is a `do` expression.

  For example, the following code is accepted by GHC:

```
main = do args <- getArgs
          if null args then return [] else do
          ps <- mapM process args
          mapM print ps
```

  This behaviour is controlled by the *NondecreasingIndentation* (page 733) extension.

NondecreasingIndentation

Since 7.2.1

Status Included in *Haskell98* (page 272)

Allow nested contexts to be at the same indentation level as its enclosing context.

- GHC doesn't do the fixity resolution in expressions during parsing as required by Haskell 98 (but not by Haskell 2010). For example, according to the Haskell 98 report, the following expression is legal:

```
let x = 42 in x == 42 == True
```

  and parses as:

```
(let x = 42 in x == 42) == True
```

  because according to the report, the `let` expression "extends as far to the right as possible". Since it can't extend past the second equals sign without causing a parse error (== is non-fix), the `let`-expression must terminate there. GHC simply gobbles up the whole expression, parsing like this:

```
(let x = 42 in x == 42 == True)
```

16.1.1.3 Expressions and patterns

By default, GHC makes some programs slightly more defined than they should be. For example, consider

```
f :: [a] -> b -> b
f [] = error "urk"
f (x:xs) = \v -> v

main = print (f [] `seq` True)
```

This should call `error` but actually prints `True`. Reason: GHC eta-expands `f` to

```
f :: [a] -> b -> b
f []     v = error "urk"
f (x:xs) v = v
```

For most programs this improves efficiency enough to be enabled & bad only in few rare cases. To suppress this optimisation use *-fpedantic-bottoms* (page 121).

16.1.1.4 Failable patterns

Since the MonadFail Proposal (MFP), do-notation blocks that contain a failable pattern need a MonadFail constraint.

For example

```
mayFail :: (MonadIO m) => m ()
mayFail = do
  (Just value) <- fetchData
  putStrLn value
```

Will warn you with

```
• Could not deduce (MonadFail m)
    arising from a do statement
    with the failable pattern '(Just x)'
  from the context: MonadIO m
    bound by the type signature for:
               mayFail :: forall (m :: * -> *). MonadIO m => m ()
```

And indeed, since the Monad class does not have the `fail` method anymore, we need to explicitly add `(MonadFail m)` to the constraints of the function.

16.1.1.5 Typechecking of recursive binding groups

The Haskell Report specifies that a group of bindings (at top level, or in a `let` or `where`) should be sorted into strongly-connected components, and then type-checked in dependency order (Haskell Report, Section 4.5.1). As each group is type-checked, any binders of the group that have an explicit type signature are put in the type environment with the specified polymorphic type, and all others are monomorphic until the group is generalised (Haskell Report, Section 4.5.2).

Following a suggestion of Mark Jones, in his paper Typing Haskell in Haskell, GHC implements a more general scheme. In GHC *the dependency analysis ignores references to variables that have an explicit type signature*. As a result of this refined dependency analysis, the dependency groups are smaller, and more bindings will typecheck. For example, consider:

```
f :: Eq a => a -> Bool
f x = (x == x) || g True || g "Yes"

g y = (y <= y) || f True
```

This is rejected by Haskell 98, but under Jones's scheme the definition for `g` is typechecked first, separately from that for `f`, because the reference to `f` in `g`'s right hand side is ignored by the dependency analysis. Then `g`'s type is generalised, to get

```
g :: Ord a => a -> Bool
```

Now, the definition for `f` is typechecked, with this type for `g` in the type environment.

The same refined dependency analysis also allows the type signatures of mutually-recursive functions to have different contexts, something that is illegal in Haskell 98 (Section 4.5.2, last sentence). GHC only insists that the type signatures of a *refined* group have identical type signatures; in practice this means that only variables bound by the same pattern binding must have the same context. For example, this is fine:

```
f :: Eq a => a -> Bool
f x = (x == x) || g True

g :: Ord a => a -> Bool
g y = (y <= y) || f True
```

16.1.1.6 Default Module headers with -main-is

The Haskell2010 Report specifies in <https://www.haskell.org/onlinereport/haskell2010/haskellch5.html#x11-990005.1> that

> "**An abbreviated form of module, consisting only of the module body,** is permitted. If this is used, the header is assumed to be *module Main(main) where*."

GHC's `-main-is` option can be used to change the name of the top-level entry point from `main` to any other variable. When compiling the main module and `-main-is` has been used to rename the default entry point, GHC will also use the alternate name in the default export list.

Consider the following program:

```
-- file: Main.hs
program :: IO ()
program = return ()
```

GHC will successfully compile this module with `ghc -main-is Main.program Main.hs`, because the default export list will include `program` rather than `main`, as the Haskell Report typically requires.

This change only applies to the main module. Other modules will still export `main` from a default export list, regardless of the `-main-is` flag. This allows use of `-main-is` with existing modules that export `main` via a default export list, even when `-main-is` points to a different entry point, as in this example (compiled with `-main-is MainWrapper.program`).

```
-- file MainWrapper.hs
module MainWrapper where
import Main

program :: IO ()
program = putStrLn "Redirecting..." >> main

-- file Main.hs
main :: IO ()
main = putStrLn "I am main."
```

16.1.1.7 Module system and interface files

GHC requires the use of `hs-boot` files to cut the recursive loops among mutually recursive modules as described in *How to compile mutually recursive modules* (page 207). This more of an infelicity than a bug: the Haskell Report says (Section 5.7)

> "Depending on the Haskell implementation used, separate compilation of mutually recursive modules may require that imported modules contain additional information so that they may be referenced before they are compiled. Explicit type signatures for all exported values may be necessary to deal with mutual recursion. The precise details of separate compilation are not defined by this Report."

16.1.1.8 Numbers, basic types, and built-in classes

`Num` superclasses The `Num` class does not have `Show` or `Eq` superclasses.

You can make code that works with both Haskell98/Haskell2010 and GHC by:

- Whenever you make a `Num` instance of a type, also make `Show` and `Eq` instances, and
- Whenever you give a function, instance or class a `Num t` constraint, also give it `Show t` and `Eq t` constraints.

`Bits` superclass The `Bits` class does not have a `Num` superclass. It therefore does not have default methods for the `bit`, `testBit` and `popCount` methods.

You can make code that works with both Haskell 2010 and GHC by:

- Whenever you make a `Bits` instance of a type, also make a `Num` instance, and
- Whenever you give a function, instance or class a `Bits t` constraint, also give it a `Num t` constraint, and
- Always define the `bit`, `testBit` and `popCount` methods in `Bits` instances.

`Read` class methods The `Read` class has two extra methods, `readPrec` and `readListPrec`, that are not found in the Haskell 2010 since they rely on the `ReadPrec` data type, which requires the *`RankNTypes`* (page 402) extension. GHC also derives `Read` instances by implementing `readPrec` instead of `readsPrec`, and relies on a default implementation of `readsPrec` that is defined in terms of `readPrec`. GHC adds these two extra methods simply because `ReadPrec` is more efficient than `ReadS` (the type on which `readsPrec` is based).

`Monad` superclass The `Monad` class has an `Applicative` superclass. You cannot write `Monad` instances that work for GHC and also for a Haskell 2010 implementation that does not define `Applicative`.

Extra instances The following extra instances are defined:

```
instance Functor ((->) r)
instance Monad ((->) r)
instance Functor ((,) a)
instance Functor (Either a)
instance Monad (Either e)
```

Multiply-defined array elements not checked This code fragment should elicit a fatal error, but it does not:

```
main = print (array (1,1) [(1,2), (1,3)])
```

GHC's implementation of `array` takes the value of an array slot from the last (index,value) pair in the list, and does no checking for duplicates. The reason for this is efficiency, pure and simple.

16.1.1.9 In `Prelude` support

`splitAt` semantics `Data.List.splitAt` is more strict than specified in the Report. Specifically, the Report specifies that

```
splitAt n xs = (take n xs, drop n xs)
```

which implies that

```
splitAt undefined undefined = (undefined, undefined)
```

but GHC's implementation is strict in its first argument, so

```
splitAt undefined [] = undefined
```

Showing records The Haskell 2010 definition of `Show` stipulates that the rendered string should only include parentheses which are necessary to unambiguously parse the result. For historical reasons, `Show` instances derived by GHC include parentheses around records despite the fact that record syntax binds more tightly than function application; e.g.,

```
data Hello = Hello { aField :: Int } deriving (Show)

-- GHC produces...
show (Just (Hello {aField=42})) == "Just (Hello {aField=42})"

-- whereas Haskell 2010 calls for...
show (Just (Hello {aField=42})) == "Just Hello {aField=42}"
```

Reading integers GHC's implementation of the `Read` class for integral types accepts hexadecimal and octal literals (the code in the Haskell 98 report doesn't). So, for example,

```
read "0xf00" :: Int
```

works in GHC.

A possible reason for this is that `readLitChar` accepts hex and octal escapes, so it seems inconsistent not to do so for integers too.

`isAlpha` The Haskell 98 definition of `isAlpha` is:

```
isAlpha c = isUpper c || isLower c
```

GHC's implementation diverges from the Haskell 98 definition in the sense that Unicode alphabetic characters which are neither upper nor lower case will still be identified as alphabetic by `isAlpha`.

`hGetContents` Lazy I/O throws an exception if an error is encountered, in contrast to the Haskell 98 spec which requires that errors are discarded (see Section 21.2.2 of the Haskell 98 report). The exception thrown is the usual IO exception that would be thrown if the failing IO operation was performed in the IO monad, and can be caught by `System.IO.Error.catch` or `Control.Exception.catch`.

16.1.1.10 The Foreign Function Interface

`hs_init()`, `hs_exit()` The FFI spec requires the implementation to support re-initialising itself after being shut down with `hs_exit()`, but GHC does not currently support that. See #13693.

16.1.2 GHC's interpretation of undefined behaviour in Haskell 98 and Haskell 2010

This section documents GHC's take on various issues that are left undefined or implementation specific in Haskell 98.

`Char` Following the ISO-10646 standard, `maxBound :: Char` in GHC is `0x10FFFF`.

`Int` In GHC the `Int` type follows the size of an address on the host architecture; in other words it holds 32 bits on a 32-bit machine, and 64-bits on a 64-bit machine.

Arithmetic on `Int` is unchecked for overflow`Int`, so all operations on `Int` happen modulo $2^{\langle n \rangle}$ where ⟨n⟩ is the size in bits of the `Int` type.

The `fromInteger` (and hence also `fromIntegral`) is a special case when converting to `Int`. The value of `fromIntegral x :: Int` is given by taking the lower ⟨n⟩ bits of `(abs x)`, multiplied by the sign of `x` (in 2's complement ⟨n⟩-bit arithmetic). This behaviour was chosen so that for example writing `0xffffffff :: Int` preserves the bit-pattern in the resulting `Int`.

Negative literals, such as `-3`, are specified by (a careful reading of) the Haskell Report as meaning `Prelude.negate (Prelude.fromInteger 3)`. So `-2147483648` means `negate (fromInteger 2147483648)`. Since `fromInteger` takes the lower 32 bits of the representation, `fromInteger (2147483648::Integer)`, computed at type `Int` is `-2147483648::Int`. The `negate` operation then overflows, but it is unchecked, so `negate (-2147483648::Int)` is just `-2147483648`. In short, one can write `minBound::Int` as a literal with the expected meaning (but that is not in general guaranteed).

The `fromIntegral` function also preserves bit-patterns when converting between the sized integral types (`Int8`, `Int16`, `Int32`, `Int64` and the unsigned `Word` variants), see the modules `Data.Int` and `Data.Word` in the library documentation.

Unchecked floating-point arithmetic Operations on `Float` and `Double` numbers are *unchecked* for overflow, underflow, and other sad occurrences. (note, however, that some architectures trap floating-point overflow and loss-of-precision and report a floating-point exception, probably terminating the program)

Large tuple support The Haskell Report only requires implementations to provide tuple types and their accompanying standard instances up to size 15. GHC limits the size of tuple types to 62 and provides instances of `Eq`, `Ord`, `Bounded`, `Read`, `Show`, and `Ix` for tuples up to size 15.

16.2 Known bugs or infelicities

The bug tracker lists bugs that have been reported in GHC but not yet fixed: see the GHC issue tracker. In addition to those, GHC also has the following known bugs or infelicities. These bugs are more permanent; it is unlikely that any of them will be fixed in the short term.

16.2.1 Bugs in GHC

- GHC's runtime system implements cooperative multitasking, with context switching potentially occurring only when a program allocates. This means that programs that do not allocate may never context switch. This is especially true of programs using STM, which may deadlock after observing inconsistent state. See #367 for further discussion.

  If you are hit by this, you may want to compile the affected module with *-fno-omit-yields* (page 121) (see *-f*: platform-independent flags* (page 113)). This flag ensures that yield points are inserted at every function entrypoint (at the expense of a bit of performance).

- GHC does not allow you to have a data type with a context that mentions type variables that are not data type parameters. For example:

  ```
  data C a b => T a = MkT a
  ```

  so that MkT's type is

  ```
  MkT :: forall a b. C a b => a -> T a
  ```

  In principle, with a suitable class declaration with a functional dependency, it's possible that this type is not ambiguous; but GHC nevertheless rejects it. The type variables mentioned in the context of the data type declaration must be among the type parameters of the data type.

- GHC's inliner can be persuaded into non-termination using the standard way to encode recursion via a data type:

  ```
  data U = MkU (U -> Bool)

  russel :: U -> Bool
  russel u@(MkU p) = not $ p u

  x :: Bool
  x = russel (MkU russel)
  ```

  The non-termination is reported like this:

  ```
  ghc: panic! (the 'impossible' happened)
    (GHC version 8.2.1 for x86_64-unknown-linux):
      Simplifier ticks exhausted
    When trying UnfoldingDone x_alB
    To increase the limit, use -fsimpl-tick-factor=N (default 100)
  ```

  with the panic being reported no matter how high a *-fsimpl-tick-factor* (page 122) you supply.

  We have never found another class of programs, other than this contrived one, that makes GHC diverge, and fixing the problem would impose an extra overhead on every compilation. So the bug remains un-fixed. There is more background in Secrets of the GHC inliner.

- On 32-bit x86 platforms when using the native code generator, the *-fexcess-precision* (page 117) option is always on. This means that floating-point calculations are non-deterministic, because depending on how the program is compiled (optimisation settings, for example), certain calculations might be done at 80-bit precision instead of the intended 32-bit or 64-bit precision. Floating-point results may differ when optimisation is turned on. In the worst case, referential transparency is violated, because for example `let x = E1 in E2` can evaluate to a different value than `E2[E1/x]`.

  One workaround is to use the *-msse2* (page 82) option (see *Platform-specific Flags* (page 82)), which generates code to use the SSE2 instruction set instead of the x87 instruction set. SSE2 code uses the correct precision for all floating-point operations, and so gives deterministic results. However, note that this only works with processors that support SSE2 (Intel Pentium 4 or AMD Athlon 64 and later), which is why the option is not enabled by default. The libraries that come with GHC are probably built without this option, unless you built GHC yourself.

- The *state hack* (page 121) optimization can result in non-obvious changes in evaluation ordering which may hide exceptions, even with *-fpedantic-bottoms* (page 121) (see, e.g., #7411). For instance,

```
import Control.Exception
import Control.DeepSeq
main = do
    evaluate (('a' : undefined) `deepseq` return () :: IO ())
    putStrLn "Hello"
```

  Compiling this program with `-O` results in `Hello` to be printed, despite the fact that `evaluate` should have bottomed. Compiling with `-O -fno-state-hack` results in the exception one would expect.

- Programs compiled with *-fdefer-type-errors* (page 414) may fail a bit more eagerly than one might expect. For instance,

```
{-# OPTIONS_GHC -fdefer-type-errors #-}
main = do
  putStrLn "Hi there."
  putStrLn True
```

  Will emit no output, despite the fact that the ill-typed term appears after the well-typed `putStrLn "Hi there."`. See #11197.

- Despite appearances `*` and `Constraint` aren't really distinct kinds in the compiler's internal representation and can be unified producing unexpected results. See #11715 for one example.

- Because of a toolchain limitation we are unable to support full Unicode paths on Windows. On Windows we support up to Latin-1. See #12971 for more.

16.2.2 Bugs in GHCi (the interactive GHC)

- GHCi does not respect the `default` declaration in the module whose scope you are in. Instead, for expressions typed at the command line, you always get the default default-type behaviour; that is, `default(Int,Double)`.

  It would be better for GHCi to record what the default settings in each module are, and use those of the 'current' module (whatever that is).

- On Windows, there's a GNU ld/BFD bug whereby it emits bogus PE object files that have more than 0xffff relocations. When GHCi tries to load a package affected by this bug, you get an error message of the form

  ```
  Loading package javavm ... linking ... WARNING: Overflown relocation field (#␣
  ↪relocs found: 30765)
  ```

  The last time we looked, this bug still wasn't fixed in the BFD codebase, and there wasn't any noticeable interest in fixing it when we reported the bug back in 2001 or so.

  The workaround is to split up the .o files that make up your package into two or more .o's, along the lines of how the `base` package does it.

CHAPTER

SEVENTEEN

EVENTLOG ENCODINGS

This section documents the encodings of the events emitted to GHC's *event log* (page 195). These events can include information about the thread scheduling events, garbage collection statistics, profiling information, user-defined tracing events.

This section is intended for implementors of tooling which consume these events. GHC ships with a C header file (`EventlogFormat.h`) which provides symbolic names for the event type IDs described in this file.

17.1 Event log format

The log format is designed to be extensible: old tools should be able to parse (but not necessarily understand all of) new versions of the format, and new tools will be able to understand old log files.

- The format is endian-independent: all values are represented in big-endian order.
- The format is extensible:
  - The header describes each event type and its length. Tools that don't recognise a particular event type can skip those events.
  - There is room for extra information in the event type specification, which can be ignored by older tools.
  - Events can have extra information added, but existing fields cannot be changed. Tools should ignore extra fields at the end of the event record.

The event-log stream begins with a header describing the event types (`EventType`) present in the file. The header is followed by the event records (`Event`) themselves, each of which start with the event type id and a 64-bit timestamp:

```
EventLog :
      EVENT_HEADER_BEGIN
      EVENT_HET_BEGIN -- header event types begin
      EventType*
      EVENT_HET_END -- header event types end
      EVENT_HEADER_END
      EVENT_DATA_BEGIN
      Event*
      EVENT_DATA_END

EventType :
      EVENT_ET_BEGIN
```

(continues on next page)

(continued from previous page)

```
     Word16         -- event type id, unique identifier for this event
     Int16          -- >=0  size of the event record in bytes (minus the event type␣
↪id and timestamp fields)
                    -- -1   variable size
     Word32         -- size of the event description in bytes
     Word8*         -- event description, UTF8 encoded string describing the event
     Word32         -- size of the extra info in bytes
     Word8*         -- extra info (for future extensions)
     EVENT_ET_END

Event :
     Word16         -- event type id, as included in the event log header
     Word64         -- timestamp (nanoseconds)
     [Word16]       -- length of the rest (optional, for variable-sized events only)
     ... event specific info ...
```

There are two classes of event types:

- *Fixed size*: All event records of a fixed-sized type are of the same length, the size given in the header event-log header.
- *Variable size*: Each event record includes a length field.

17.2 Runtime system diagnostics

- `ThreadId ~ Word32`
- `CapNo ~ Word16`
- `CapSetId ~ Word32`

17.2.1 Capability sets

TODO

17.2.2 Environment information

These events are typically produced during program startup and describe the environment which the program is being run in.

`RTS_IDENTIFIER`

> **Tag** 29
>
> **Length** variable
>
> **Field CapSetId** Capability set
>
> **Field String** Runtime system name and version.

Describes the name and version of the runtime system responsible for the indicated capability set.

`PROGRAM_ARGS`

> **Tag** 30

Length variable

Field CapSetId Capability set

Field [String] The command-line arguments passed to the program

Describes the command-line used to start the program.

PROGRAM_ENV

Tag 31

Length variable

Field CapSetId Capability set

Field [String] The environment variable name/value pairs. (TODO: encoding?)

Describes the environment variables present in the program's environment.

WALL_CLOCK_TIME

Tag 43

Length fixed

Field CapSetId Capability set

Field Word64 Unix epoch seconds

Field Word32 Nanoseconds

Records the wall clock time to make it possible to correlate events from elsewhere with the eventlog.

17.2.3 Thread and scheduling events

CREATE_THREAD

Tag 0

Length fixed

Field ThreadId thread id

Marks the creation of a Haskell thread.

RUN_THREAD

Tag 1

Length fixed

Field ThreadId thread id

The indicated thread has started running.

STOP_THREAD

Tag 2

Length fixed

Field ThreadId thread id

Field Word16 status

- 1: HeapOverflow

- 2: StackOverflow
- 3: ThreadYielding
- 4: ThreadBlocked
- 5: ThreadFinished
- 6: ForeignCall
- 7: BlockedOnMVar
- 8: BlockedOnBlackHole
- 9: BlockedOnRead
- 10: BlockedOnWrite
- 11: BlockedOnDelay
- 12: BlockedOnSTM
- 13: BlockedOnDoProc
- 16: BlockedOnMsgThrowTo
- 20: BlockedOnMVarRead

Field ThreadId thread id of thread being blocked on (only for some status values)

The indicated thread has stopped running for the reason given by `status`.

`THREAD_RUNNABLE`

Tag 3

Length fixed

Field ThreadId thread id

The indicated thread is has been marked as ready to run.

`MIGRATE_THREAD`

Tag 4

Length fixed

Field ThreadId thread id

Field CapNo capability

The indicated thread has been migrated to a new capability.

`THREAD_WAKEUP`

Tag 8

Length fixed

Field ThreadId thread id

Field CapNo other capability

The indicated thread has been woken up on another capability.

`THREAD_LABEL`

Tag 44

Length variable

Field ThreadId thread id

Field String label

The indicated thread has been given a label (e.g. with GHC.Conc.labelThread).

17.2.4 Garbage collector events

The following events mark various points of the lifecycle of a moving garbage collection.

A typical garbage collection will look something like the following:

1. A capability realizes that it needs a garbage collection (e.g. as a result of running out of nursery) and requests a garbage collection. This is marked by *REQUEST_SEQ_GC* (page 748) or *REQUEST_PAR_GC* (page 748).
2. As other capabilities reach yield points and suspend execution they emit *STOP_THREAD* (page 745) events.
3. When all capabilities have suspended execution, collection will begin, marked by a *GC_START* (page 747) event.
4. As individual parallel GC threads commence with scavenging they will emit *GC_WORK* (page 748) events.
5. If a parallel GC thread runs out of work it will emit a *GC_IDLE* (page 748) event. If it is later handed more work it will emit another *GC_WORK* (page 748) event.
6. Eventually when scavenging has finished a *GC_DONE* (page 748) event will be emitted by each GC thread.
7. A bit of book-keeping is performed.
8. A *GC_END* (page 748) event will be emitted marking the end of the GC cycle.
9. A *HEAP_SIZE* (page 749) event will be emitted giving the current size of the heap, in bytes, calculated by how many megablocks are allocated.
10. A *BLOCKS_SIZE* (page 749) event will be emitted giving the current size of the heap, in bytes, calculated by how many blocks are allocated.
11. A *GC_STATS_GHC* (page 748) event will be emitted containing various details of the collection and heap state.
12. In the case of a major collection, a *HEAP_LIVE* (page 750) event will be emitted describing the current size of the live on-heap data.
13. In the case of the *-threaded* (page 248) RTS, a *SPARK_COUNTERS* (page 750) event will be emitted giving details on how many sparks have been created, evaluated, and GC'd.
14. As mutator threads resume execution they will emit *RUN_THREAD* (page 745) events.
15. A *MEM_RETURN* (page 749) event will be emitted containing details about currently live mblocks, how many we think we need and whether we could return excess to the OS.

Note that in the case of the concurrent non-moving collector additional events will be emitted during the concurrent phase of collection. These are described in *Non-moving GC event output* (page 757).

GC_START

Tag 9

Length fixed

A garbage collection pass has been started.

GC_END

Tag 10

Length fixed

A garbage collection pass has been finished.

REQUEST_SEQ_GC

Tag 11

Length fixed

A sequential garbage collection has been requested by a capability.

REQUEST_PAR_GC

Tag 12

Length fixed

A parallel garbage collection has been requested by a capability.

GC_IDLE

Tag 20

Length fixed

An idle-time garbage collection has been started.

GC_WORK

Tag 21

Length fixed

Marks the start of concurrent scavenging.

GC_DONE

Tag 22

Length fixed

Marks the end of concurrent scavenging.

GC_STATS_GHC

Tag 53

Length fixed

Field CapSetId heap capability set

Field Word16 generation of collection

Field Word64 bytes copied

Field Word64 bytes of slop found

Field Word64 bytes of fragmentation, the difference between total mblock size and total block size. When all mblocks are full of full blocks, this number is 0.

Field Word32 number of parallel garbage collection threads

Field Word64 maximum number of bytes copied by any single collector thread

Field Word64 total bytes copied by all collector threads

Field Word64 the amount of balanced data copied by all threads

Report various information about a major collection.

GC_GLOBAL_SYNC

Tag 54

Length fixed

TODO

MEM_RETURN

Tag 90

Length fixed

Field CapSetId heap capability set

Field Word32 currently allocated mblocks

Field Word32 the number of mblocks we would like to retain

Field Word32 the number of mblocks which we returned to the OS

Report information about currently allocation megablocks and attempts made to return them to the operating system. If your heap is fragmented then the current value will be greater than needed value but returned will be less than the difference between the two.

17.2.5 Heap events and statistics

HEAP_ALLOCATED

Tag 49

Length fixed

Field CapSetId heap capability set

Field Word64 allocated bytes

A new chunk of heap has been allocated by the indicated capability set.

HEAP_SIZE

Tag 50

Length fixed

Field CapSetId heap capability set

Field Word64 heap size in bytes

Report the heap size, calculated by the number of megablocks currently allocated.

BLOCKS_SIZE

Tag 91

Length fixed

Field CapSetId heap capability set

Field Word64 heap size in bytes

Report the heap size, calculated by the number of blocks currently allocated.

HEAP_LIVE

Tag 51

Length fixed

Field CapSetId heap capability set

Field Word64 heap size in bytes

Report the live heap size.

HEAP_INFO_GHC

Tag 52

Length fixed

Field CapSetId heap capability set

Field Word16 number of garbage collection generations

Field Word64 maximum heap size

Field Word64 allocation area size

Field Word64 MBlock size

Field Word64 Block size

Report various information about the heap configuration. Typically produced during RTS initialization..

17.2.6 Spark events

CREATE_SPARK_THREAD

Tag 15

Length fixed

A thread has been created to perform spark evaluation.

SPARK_COUNTERS

Tag 34

Length fixed

A periodic reporting of various statistics of spark evaluation.

SPARK_CREATE

Tag 35

Length fixed

A spark has been added to the spark pool.

SPARK_DUD

Tag 36

Length fixed

TODO

SPARK_OVERFLOW

Tag 37

Length fixed

TODO

SPARK_RUN

Tag 38

Length fixed

Evaluation has started on a spark.

SPARK_STEAL

Tag 39

Length fixed

Field Word16 capability from which the spark was stolen

A spark has been stolen from another capability for evaluation.

SPARK_FIZZLE

Tag 40

Length fixed

A spark has been GC'd before being evaluated.

SPARK_GC

Tag 41

Length fixed

An unevaluated spark has been garbage collected.

17.2.7 Capability events

CAP_CREATE

Tag 45

Length fixed

Field CapNo the capability number

A capability has been started.

CAP_DELETE

Tag 46

Length fixed

A capability has been deleted.

CAP_DISABLE

Tag 47

Length fixed

A capability has been disabled.

`CAP_ENABLE`

Tag 48

Length fixed

A capability has been enabled.

17.2.8 Task events

`TASK_CREATE`

Tag 55

Length fixed

Field TaskId task id

Field CapNo capability number

Field KernelThreadId The thread-id of the kernel thread which created the task.

Marks the creation of a task.

`TASK_MIGRATE`

Tag 56

Length fixed

Field TaskId task id

Field CapNo old capability

Field CapNo new capability

Marks the migration of a task to a new capability.

`TASK_DELETE`

Tag 57

Length fixed

Field TaskId task id

Marks the deletion of a task.

17.2.9 Tracing events

`LOG_MSG`

Tag 16

Length variable

Field String The message

A log message from the runtime system.

`BLOCK_MARKER`

Tag 18

Length fixed

Field Word32 block size

Field Word64 end time in nanoseconds

Field Word16 capability number, invalid if `0xffff`.

Marks a chunk of events. The events that fit in the next `block size` bytes all belong to the block marker capability.

USER_MSG

Tag 19

Length variable

Field String message

A user log message (from, e.g., Control.Concurrent.traceEvent).

USER_MARKER

Tag 58

Length variable

Field String marker name

A user marker (from Debug.Trace.traceMarker).

17.3 Heap profiler event log output

The heap profiler can produce output to GHC's event log, allowing samples to be correlated with other event log events over the program's lifecycle.

This section defines the layout of these events. The `String` type below is defined to be a UTF-8 encoded NUL-terminated string.

17.3.1 Metadata event types

17.3.1.1 Beginning of sample stream

A single fixed-width event emitted during program start-up describing the samples that follow.

HEAP_PROF_BEGIN

Tag 160

Length variable

Field Word8 profile ID

Field Word64 sampling period in nanoseconds

Field Word32 sample breadown type. One of,

- `HEAP_PROF_BREAKDOWN_COST_CENTER` (output from *-hc* (page 665))
- `HEAP_PROF_BREAKDOWN_CLOSURE_DESCR` (output from *-hd* (page 665))
- `HEAP_PROF_BREAKDOWN_RETAINER` (output from *-hr* (page 666))
- `HEAP_PROF_BREAKDOWN_MODULE` (output from *-hm* (page 665))

- `HEAP_PROF_BREAKDOWN_TYPE_DESCR` (output from *-hy* (page 665))
- `HEAP_PROF_BREAKDOWN_BIOGRAPHY` (output from *-hb* (page 666))
- `HEAP_PROF_BREAKDOWN_CLOSURE_TYPE` (output from *-hT* (page 194))

Field String module filter

Field String closure description filter

Field String type description filter

Field String cost centre filter

Field String cost centre stack filter

Field String retainer filter

Field String biography filter

17.3.1.2 Cost centre definitions

A variable-length packet produced once for each cost centre,

`HEAP_PROF_COST_CENTRE`

Tag 161

Length fixed

Field Word32 cost centre number

Field String label

Field String module

Field String source location

Field Word8 flags:

- bit 0: is the cost-centre a CAF?

17.3.1.3 Info Table Provenance definitions

A message which describes an approximate source position for info tables. See *-finfo-table-map* (page 687) for more information.

`IPE`

Tag 169

Length fixed

Field Word64 info table address

Field String table name

Field String closure type

Field String type

Field String source position label

Field String source position module

Field String source position location

17.3.1.4 Sample event types

A sample (consisting of a list of break-down classes, e.g. cost centres, and heap residency sizes), is to be encoded in the body of one or more events.

We normally mark the beginning of a new sample with an `EVENT_HEAP_PROF_SAMPLE_BEGIN` event,

`HEAP_PROF_SAMPLE_BEGIN`

> **Length** fixed
>
> **Field Word64** sample number
>
> Marks the beginning of a heap profile sample.

Biographical profiling samples start with the `EVENT_HEAP_BIO_PROF_SAMPLE_BEGIN` event. These events also include a timestamp which indicates when the sample was taken. This is because all these samples will appear at the end of the eventlog due to how the biographical profiling mode works. You can use the timestamp to reorder the samples relative to the other events.

`HEAP_BIO_PROF_SAMPLE_BEGIN`

> **Tag** 166
>
> **Length** fixed
>
> **Field Word64** sample number
>
> **Field Word64** eventlog timestamp in ns

A heap residency census will follow. Since events may only be up to 2^16^ bytes in length a single sample may need to be split among multiple `EVENT_HEAP_PROF_SAMPLE` events. The precise format of the census entries is determined by the break-down type.

At the end of the sample period the `EVENT_HEAP_PROF_SAMPLE_END` event if emitted. This is useful to properly delimit the sampling period and to record the total time spent profiling.

`HEAP_PROF_SAMPLE_END`

> **Tag** 165
>
> **Length** fixed
>
> **Field Word64** sample number
>
> Marks the end of a heap profile sample.

17.3.1.5 Cost-centre break-down

A variable-length packet encoding a heap profile sample broken down by,

- cost-centre (*`-hc`* (page 665))

`HEAP_PROF_SAMPLE_COST_CENTRE`

> **Tag** 163
>
> **Length** variable
>
> **Field Word8** profile ID
>
> **Field Word64** heap residency in bytes
>
> **Field Word8** stack depth

Field Word32[] cost centre stack starting with inner-most (cost centre numbers)

17.3.1.6 String break-down

A variable-length event encoding a heap sample broken down by,

- type description (*-hy* (page 665))
- closure description (*-hd* (page 665))
- module (*-hm* (page 665))

HEAP_PROF_SAMPLE_STRING

Tag 164

Length variable

Field Word8 profile ID

Field Word64 heap residency in bytes

Field String type or closure description, or module name

17.4 Time profiler event log output

The time profiling mode enabled by *-p* (page 659) also emits sample events to the eventlog. At the start of profiling the tick interval is emitted to the eventlog and then on each tick the current cost centre stack is emitted. Together these enable a user to construct an approximate track of the executation of their program.

17.4.1 Profile begin event

PROF_BEGIN

Tag 168

Length fixed

Field Word64 tick interval, in nanoseconds

Marks the beginning of a time profile.

17.4.2 Profile sample event

A variable-length packet encoding a profile sample.

PROF_SAMPLE_COST_CENTRE

Tag 167

Length variable

Field Word32 capability

Field Word64 current profiling tick

Field Word8 stack depth

Field Word32[] cost centre stack starting with inner-most (cost centre numbers)

17.5 Biographical profile sample event

A variable-length packet encoding a profile sample.

BIO_PROF_SAMPLE_BEGIN

Tag 166

TODO

17.6 Non-moving GC event output

These events mark various stages of the *non-moving collection* (page 184) lifecycle. These are enabled with the `+RTS -lg` event-set.

A typical non-moving collection cycle will look something like the following:

1. The preparatory phase of collection will emit the usual events associated with a moving collection. See *Garbage collector events* (page 747) for details.
2. The concurrent write barrier is enabled and the concurrent mark thread is started. From this point forward mutator threads may emit *CONC_UPD_REM_SET_FLUSH* (page 758) events, indicating that they have flushed their capability-local update remembered sets.
3. Concurrent marking begins, denoted by a *CONC_MARK_BEGIN* (page 757) event.
4. When the mark queue is depleted a *CONC_MARK_END* (page 758) is emitted.
5. If necessary (e.g. due to weak pointer marking), the marking process will continue, returning to step (3) above.
6. When the collector has done as much concurrent marking as it can it will enter the post-mark synchronization phase of collection, denoted by a *CONC_SYNC_BEGIN* (page 758) event.
7. Mutator threads will suspend execution and, if necessary, flush their update remembered sets (indicated by *CONC_UPD_REM_SET_FLUSH* (page 758) events).
8. The collector will do any final marking necessary (indicated by *CONC_MARK_BEGIN* (page 757) and *CONC_MARK_END* (page 758) events).
9. The collector will do a small amount of sweeping, disable the write barrier, emit a *CONC_SYNC_END* (page 758) event, and allow mutators to resume
10. The collector will begin the concurrent sweep phase, indicated by a *CONC_SWEEP_BEGIN* (page 758) event.
11. Once sweeping has concluded a *CONC_SWEEP_END* (page 758) event will be emitted and the concurrent collector thread will terminate.
12. A *NONMOVING_HEAP_CENSUS* (page 758) event will be emitted describing the fragmentation state of the non-moving heap.

CONC_MARK_BEGIN

Tag 200

Length fixed

Marks the beginning of marking by the concurrent collector.

CONC_MARK_END

Tag 201

Length fixed

Field Word32 number of objects which were marked in this marking phase.

Marks the end of marking by the concurrent collector.

CONC_SYNC_BEGIN

Tag 202

Length fixed

Marks the beginning of the concurrent garbage collector's post-mark synchronization phase.

CONC_SYNC_END

Tag 203

Length fixed

Marks the end of the concurrent garbage collector's post-mark synchronization phase.

CONC_SWEEP_BEGIN

Tag 204

Length fixed

Marks the beginning of the concurrent garbage collector's sweep phase.

CONC_SWEEP_END

Tag 205

Length fixed

Marks the end of the concurrent garbage collector's sweep phase.

CONC_UPD_REM_SET_FLUSH

Tag 206

Length fixed

Marks a capability flushing its local update remembered set accumulator.

17.6.1 Non-moving heap census

The non-moving heap census events (enabled with the `+RTS -ln` (page 195) event-set) are intended to provide insight into fragmentation of the non-moving heap.

NONMOVING_HEAP_CENSUS

Tag 207

Length fixed

Field Word16 *blk_sz* in bytes.

Field Word32 number of active segments.

Field Word32 number of filled segments.

Field Word32 number of live blocks.

Describes the occupancy of the *blk_sz* sub-heap.

17.6.2 Ticky counters

Programs compiled with *-ticky* (page 677) and *-eventlog* (page 248) and invoked with *+RTS -lT* (page 195) will emit periodic samples of the ticky entry counters to the eventlog.

TICKY_COUNTER_DEF

Tag 210

Length variable

Field Word64 counter ID

Field Word16 arity/field count

Field String argument kinds. This is the same as the synonymous field in the textual ticky summary.

Field String counter name

Defines a ticky counter.

TICKY_COUNTER_BEGIN_SAMPLE

Tag 212

Length fixed

Denotes the beginning of an atomic set of ticky-ticky profiler counter samples.

TICKY_COUNTER_SAMPLE

Tag 211

Length fixed

Field Word64 counter ID

Field Word64 number of times closures of this type has been entered.

Field Word64 number of allocations (words)

Field Word64 number of times this has been allocated (words). Only produced for modules compiled with *-ticky-allocd* (page 678).

Records the number of "ticks" recorded by a ticky-ticky counter single the last sample.

CHAPTER

EIGHTEEN

GLOSSARY

technology preview GHC will occassionally ship features advertised as being in a *technology preview* state. Such features are generally opt-in in nature (e.g. new language extensions) and may have various shortcomings. These may include known bugs (which we will try to document), lacking optimisation, and unhandled interactions with other language features. As such, behavior of such features may change in the future. However, we do expect features to converge to non-preview state over the course of a few GHC major releases.

CHAPTER

NINETEEN

CARE AND FEEDING OF YOUR GHC USER'S GUIDE

The GHC User's Guide is the primary reference documentation for the Glasgow Haskell Compiler. Even more than this, it at times serves (for better or for worse) as a de-facto language standard, being the sole non-academic reference for many widely used language extensions.

Since GHC 8.0, the User's Guide is authored in reStructuredText (or reST or RST, for short) a rich but light-weight mark-up language aimed at producing documentation. The Sphinx tool is used to produce the final PDF and HTML documentation.

This document (also written in reST) serves as a brief introduction to reST and to document the conventions used in the User's Guide. This document is *not* intended to be a thorough guide to reST. For this see the resources referenced *below* (page 769).

19.1 Basics

Unicode characters are allowed in the document.

The basic syntax works largely as one would expect. For instance,

```
This is a paragraph containing a few sentences of text. Purple turtles walk
through green fields of lofty maize. Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Some lists,

1. This is a list item

   a. Followed by a sub-item
   b. And another!
   c. Now with ``a bit of code`` and some *emphasis*.

2. Back to the first list

Or perhaps you are more of a bullet list person,

* Foo
* Fizzle

  - Bar
  - Blah

Or perhaps a definition list is in order,

*Chelonii*
    The taxonomic order consisting of modern turtles
```

(continues on next page)

(continued from previous page)

```
*Meiolaniidae*
    The taxonomic order of an extinct variety of herbivorous turtles.
```

Note the blank lines between a list item and its sub-items. Sub-items should be on the same indentation level as the content of their parent items. Also note that no whitespace is necessary or desirable before the bullet or item number (lest the list be indented unnecessarily).

The above would be rendered as,

> This is a paragraph containing a few sentences of text. Purple turtles walk through green fields of lofty maize. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Some lists,
>
> 1. This is a list item
>
>   a. Followed by a sub-item
>
>   b. And another!
>
>   c. Now with `a bit of code` and some *emphasis*.
>
> 2. Back to the first list
>
> Or perhaps you are more of a bullet list person,
>
> - Foo
> - Fizzle
>   - Bar
>   - Blah
>
> Or perhaps a definition list is in order,
>
> ***Chelonii*** The taxonomic order consisting of modern turtles
>
> ***Meiolaniidae*** The taxonomic order of an extinct variety of herbivorous turtles.

19.1.1 Headings

While reST can accommodate a wide range of heading styles, we have standardized on this convention in the User's Guide,

```
Header level 1
==============

Header level 2
--------------

Header level 3
~~~~~~~~~~~~~~

Header level 4
^^^^^^^^^^^^^^
```

19.1.2 Formatting code

19.1.2.1 Haskell

Code snippets can be included as both inline and block elements. Inline code is denoted with double-backticks whereas block of code are introduced by ending a paragraph with double-colons and indentation,

```
The ``fib`` function is defined as, ::

    fib :: Integer -> Integer
    fib 1 = 1
    fib n = n * fib (n - 1)
```

Which would be rendered as,

The `fib` function is defined as,

```
fib :: Integer -> Integer
fib 1 = 1
fib n = n * fib (n - 1)
```

19.1.2.2 Other languages

Double-colon blocks are syntax-highlighted as Haskell by default. To avoid this use a `.. code-block` directive with explicit language designation,

```
This is a simple shell script,

.. code-block:: sh

    #!/bin/bash
    echo "Hello World!"
```

19.1.3 Links

19.1.3.1 Within the User's Guide

Frequently we want to give a name to a section so it can be referred to from other points in the document,

```
.. _options-platform:

Platform-specific Flags
-----------------------

There are lots of platform-specific flags.

Some other section
-------------------

GHC supports a variety of :ref:`x86 specific features <options-platform>`.

See :ref:`options-platform` for details.
```

19.1.3.2 To GHC resources

There are special macros for conveniently linking to GHC Wiki articles and tickets,

```
See :ghc-wiki:`commentary/compiler/demand` for details on demand analysis.

See the :ghc-wiki:`coding style <commentary/coding-style>` for guidelines.

See the :ghc-ticket:`123` for further discussion.

See the :ghc-ticket:`this bug <123>` for what happens when this fails.
```

19.1.3.3 To external resources

External links can be written in either of these ways,

```
See the `GHC Wiki <https://gitlab.haskell.org/ghc/ghc/wikis>`_ for details.

See the `GHC Wiki`_ for details.

.. _GHC Wiki: https://gitlab.haskell.org/ghc/ghc/wikis
```

19.1.3.4 To core library Haddock documentation

It is often useful to be able to refer to the Haddock documentation of the libraries shipped with GHC. The users guide's build system provides commands for referring to documentation for the following core GHC packages,

- `base`: `:base-ref:`
- `cabal`: `:cabal-ref:`
- `ghc-prim`: `:ghc-prim-ref:`

These are defined in `docs/users_guide/ghc_config.py.in`.

For instance,

```
See the documentation for :base-ref:`Control.Applicative.`
for details.
```

19.1.3.5 Math

You can insert type-set equations using `:math:`. For instance,

```
Fick's law of diffusion, :math:`J = -D \frac{d \varphi}{d x}`, ...
```

will render as,

> Fick's law of diffusion, $J = -D \frac{d \varphi}{d x}$, ...

19.1.4 Index entries

Index entries can be included anywhere in the document as a block element. They look like,

```
Here is some discussion on the Strict Haskell extension.

.. index::
    single: strict haskell
    single: language extensions; StrictData
```

This would produce two entries in the index referring to the "Strict Haskell" section. One would be a simple "strict haskell" heading whereas the other would be a "StrictData" sub-heading under "language extensions".

Sadly it is not possible to use inline elements (e.g. monotype inlines) inside index headings.

19.2 Citations

Citations can be marked-up like this,

```
See the original paper [Doe2008]_

.. [Doe2008] John Doe and Leslie Conway.
             "This is the title of our paper" (2008)
```

19.3 Admonitions

Admonitions are block elements used to draw the readers attention to a point. They should not be over-used for the sake of readability but they can be quite effective in separating and drawing attention to points of importance,

```
.. important::

    Be friendly and supportive to your fellow contributors.
```

Would be rendered as,

> **Important:** Be friendly and supportive to your fellow contributors.

There are a number of admonitions types,

- attention
- caution
- danger
- error
- hint
- important
- note
- tip
- warning

19.4 Documenting command-line options and GHCi commands

`conf.py` defines a few Sphinx object types for GHCi commands (`ghci-cmd`), **ghc** command-line options (`ghc-flag`), and runtime :system options (`rts-flag`),

19.4.1 Command-line options

The `ghc-flag` and `rts-flag` roles/directives can be used to document command-line arguments to the **ghc** executable and runtime system, respectively. For instance,

```
.. rts-flag:: -C ⟨seconds⟩

   :since: 8.2
   :default: 20 milliseconds

   Sets the context switch interval to ⟨s⟩ seconds.
```

Will be rendered as,

-C ⟨seconds⟩

Since 8.2

Default 20 milliseconds

Sets the context switch interval to ⟨s⟩ seconds.

and will have an associated index entry generated automatically.

The `ghc-flag` directive requires a few extra parameters to be passed. This extra information is used to generate the *Flag reference* (page 134) and the man page. A `ghc-flag` directive looks like this,

```
.. ghc-flag:: -fasm
    :shortdesc: Use the native code generator
    :type: dynamic
    :reverse: -fllvm
    :category: codegen

    Regular description...
```

When rendered, the extra parameters will be hidden, and the data stored for later use. For more details, see the Sphinx extension `flags.py`.

Note that, as in Style Conventions below, we use ⟨⟩ instead of less-than/greater-than signs. To reference a `ghc-flag` or `rts-flag`, you must match the definition exactly, including the arguments. A quick way to find the exact names and special characters is,

```
$ git grep -- "flag:: -o "
```

which will generate the appropriate,

```
separate_compilation.rst:.. ghc-flag:: -o ⟨file⟩
```

19.4.2 GHCi commands

The `ghci-cmd` role and directive can be used to document GHCi directives. For instance, we can describe the GHCi `:module` command,

```
.. ghci-cmd:: :module; [*]⟨file⟩

    Load a module
```

which will be rendered as,

> **:module** `[*]⟨file⟩`
> Load a module

And later refer to it by just the command name, `:module`,

```
The GHCi :ghci-cmd:`:load` and :ghci-cmd:`:module` commands are used
to modify the modules in scope.
```

Like command-line options, GHCi commands will have associated index entries generated automatically.

19.5 Style Conventions

When describing user commands and the like it is common to need to denote user-substitutable tokens. In this document we use the convention, `⟨subst⟩` (note that these are angle brackets, `U+27E8` and `U+27E9`, not less-than/greater-than signs).

19.6 reST reference materials

- Sphinx reST Primer: A great place to start.
- Sphinx extensions: How Sphinx extends reST
- reST reference: When you really need the details.
- Directives reference

CHAPTER

TWENTY

INDICES AND TABLES

- genindex
- search

BIBLIOGRAPHY

[AssocDataTypes2005] "Associated Types with Class", M. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. In Proceedings of "The 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)", pages 1-13, ACM Press, 2005.

[AssocTypeSyn2005] "Associated Type Synonyms", M. Chakravarty, G. Keller, and S. Peyton Jones. In Proceedings of "The Tenth ACM SIGPLAN International Conference on Functional Programming", ACM Press, pages 241-253, 2005.

[TypeFamilies2008] "Type Checking with Open Type Functions", T. Schrijvers, S. Peyton-Jones, M. Chakravarty, and M. Sulzmann, in Proceedings of "ICFP 2008: The 13th ACM SIGPLAN International Conference on Functional Programming", ACM Press, pages 51-62, 2008.

[Jones2000] "Type Classes with Functional Dependencies", Mark P. Jones, In *Proceedings of the 9th European Symposium on Programming*, ESOP 2000, Berlin, Germany, March 2000, Springer-Verlag LNCS 1782, .

[Jones1999] "Exploring the Design Space for Type-based Implicit Parameterization", Mark P. Jones, Oregon Graduate Institute of Science & Technology, Technical Report, July 1999.

[Lewis2000] "Implicit parameters: dynamic scoping with static types", J Lewis, MB Shields, E Meijer, J Launchbury, *27th ACM Symposium on Principles of Programming Languages (POPL'00)*, Boston, Jan 2000.

[Generics2010] Jose Pedro Magalhaes, Atze Dijkstra, Johan Jeuring, and Andres Loeh. A generic deriving mechanism for Haskell. Proceedings of the third ACM Haskell symposium on Haskell (Haskell'2010), pp. 37-48, ACM, 2010.

INDEX

Symbols

A

B

C

D

F

G

H

M

N

O

P

Q

R

S

U

V

W

Y

�