



GHC User's Guide Documentation

Release 8.4.0.20180118

GHC Team

January 20, 2018

1	The Glasgow Haskell Compiler License	3
2	Introduction to GHC	5
2.1	Obtaining GHC	5
2.2	Meta-information: Web sites, mailing lists, etc.	5
2.3	Reporting bugs in GHC	6
2.4	GHC version numbering policy	6
3	Release notes for version 8.4.1	9
3.1	Highlights	9
3.2	Full details	9
3.3	Included libraries	15
4	Using GHCi	17
4.1	Introduction to GHCi	17
4.2	Loading source files	18
4.3	Loading compiled code	19
4.4	Interactive evaluation at the prompt	21
4.5	The GHCi Debugger	33
4.6	Invoking GHCi	42
4.7	GHCi commands	44
4.8	The <code>:set</code> and <code>:seti</code> commands	54
4.9	The <code>.ghci</code> and <code>.haskeline</code> files	56
4.10	Compiling to object code inside GHCi	57
4.11	Running the interpreter in a separate process	58
4.12	FAQ and Things To Watch Out For	58
5	Using <code>runghc</code>	61
5.1	Usage	61
5.2	<code>runghc</code> flags	61
5.3	GHC Flags	61
6	Using GHC	63
6.1	Using GHC	63
6.2	Warnings and sanity-checking	74
6.3	Optimisation (code improvement)	87
6.4	Using Concurrent Haskell	99
6.5	Using SMP parallelism	99
6.6	Flag reference	101
6.7	Running a compiled program	131
6.8	Filenames and separate compilation	147

6.9	Packages	163
6.10	GHC Backends	180
6.11	Options related to a particular phase	181
6.12	Using shared libraries	192
6.13	Debugging the compiler	195
7	Profiling	203
7.1	Cost centres and cost-centre stacks	203
7.2	Compiler options for profiling	207
7.3	Time and allocation profiling	208
7.4	Profiling memory usage	211
7.5	hp2ps – Rendering heap profiles to PostScript	216
7.6	Profiling Parallel and Concurrent Programs	219
7.7	Observing Code Coverage	220
7.8	Using “ticky-ticky” profiling (for implementors)	225
8	Advice on: sooner, faster, smaller, thriftier	227
8.1	Sooner: producing a program more quickly	227
8.2	Faster: producing a program that runs quicker	228
8.3	Smaller: producing a program that is smaller	231
8.4	Thriftier: producing a program that gobbles less heap space	231
9	GHC Language Features	233
9.1	Language options	233
9.2	Unboxed types and primitive operations	236
9.3	Syntactic extensions	240
9.4	Extensions to data types and type synonyms	264
9.5	Extensions to the record system	277
9.6	Extensions to the “deriving” mechanism	285
9.7	Pattern synonyms	305
9.8	Class and instances declarations	312
9.9	Type families	334
9.10	Datatype promotion	349
9.11	Kind polymorphism and Type-in-Type	352
9.12	Levity polymorphism	361
9.13	Type-Level Literals	363
9.14	Constraints in types	364
9.15	Extensions to type signatures	367
9.16	Lexically scoped type variables	370
9.17	Bindings and generalisation	374
9.18	Visible type application	375
9.19	Implicit parameters	377
9.20	Arbitrary-rank polymorphism	380
9.21	Impredicative polymorphism	384
9.22	Typed Holes	385
9.23	Partial Type Signatures	387
9.24	Custom compile-time errors	392
9.25	Deferring type errors to runtime	393
9.26	Template Haskell	394
9.27	Arrow notation	404
9.28	Bang patterns and Strict Haskell	410
9.29	Assertions	417
9.30	Static pointers	418
9.31	Pragmas	420

9.32 Rewrite rules	432
9.33 Special built-in functions	438
9.34 Generic classes	439
9.35 Generic programming	439
9.36 Roles	442
9.37 HasCallStack	446
9.38 Concurrent and Parallel Haskell	447
9.39 Safe Haskell	449
10 Foreign function interface (FFI)	461
10.1 GHC differences to the FFI Chapter	461
10.2 GHC extensions to the FFI Chapter	462
10.3 Using the FFI with GHC	464
11 Extending and using GHC as a Library	473
11.1 Source annotations	473
11.2 Using GHC as a Library	474
11.3 Compiler Plugins	475
12 What to do when something goes wrong	483
12.1 When the compiler “does the wrong thing”	483
12.2 When your program “does the wrong thing”	484
13 Debugging compiled programs	485
13.1 Tutorial	485
13.2 Requesting a stack trace from Haskell code	487
13.3 Requesting a stack trace with SIGQUIT	487
13.4 Implementor’s notes: DWARF annotations	488
13.5 Further Reading	489
14 Other Haskell utility programs	491
14.1 “Yacc for Haskell”: happy	491
14.2 Writing Haskell interfaces to C code: hsc2hs	491
15 Running GHC on Win32 systems	495
15.1 Starting GHC on Windows platforms	495
15.2 Running GHCi on Windows	495
15.3 Interacting with the terminal	496
15.4 Differences in library behaviour	496
15.5 Using GHC (and other GHC-compiled executables) with Cygwin	496
15.6 Building and using Win32 DLLs	497
16 Known bugs and infelicities	501
16.1 Haskell standards vs. Glasgow Haskell: language non-compliance	501
16.2 Known bugs or infelicities	506
17 Eventlog encodings	509
17.1 Heap profiler event log output	509
18 Care and feeding of your GHC User’s Guide	511
18.1 Basics	511
18.2 Citations	515
18.3 Admonitions	515
18.4 Documenting command-line options and GHCi commands	516
18.5 Style Conventions	517

18.6 ReST reference materials	517
19 Indices and tables	519
Bibliography	521
Index	523

Contents:

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.

INTRODUCTION TO GHC

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 17), and a batch compiler, described throughout [Using GHC](#) (page 63). 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 [GHC Language Features](#) (page 233).

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 203) for more details.

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

2.1 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.

2.2 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 <http://www.haskell.org/mailman/listinfo/> for the full list.

2.3 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.

2.4 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 $\langle y \rangle$ is *even*. Releases on the stable branch $x.y$ are numbered $x.y.z$, where $\langle z \rangle$ (≥ 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 183)) for a major release $x.y.z$ is the integer $\langle xyy \rangle$ (if $\langle y \rangle$ 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.

We may make snapshot releases of the HEAD [available for download](#), and the latest sources are available from [the git repositories](#).

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 $\langle xyy \rangle$. 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 69)).

The compiler version can be tested within compiled code with the `MIN_VERSION_GLASGOW_HASKELL` CPP macro (defined only when `-XCPP` (page 183) is used). See [Standard CPP macros](#) (page 183) for details.

RELEASE NOTES FOR VERSION 8.4.1

The significant changes to the various parts of the compiler are listed in the following sections. There have also been numerous bug fixes and performance improvements over the 8.2.1 release.

Note: This compiling this release requires GCC 4.7 or newer due to [Trac #14244](#).

3.1 Highlights

The highlights, since the 8.2.1 release, are:

- GHC is now capable of deriving more instances
- More refinement of the *-XTypeInType* (page 352) story and improvements in type error messages.
- Further improvements in code generation
- Many, many bug fixes.

3.2 Full details

3.2.1 Language

- Data families have been generalised a bit: a data family declaration can now end with a kind variable *k* instead of *Type*. Additionally, data/newtype instance no longer need to list all the patterns of the family if they don't wish to; this is quite like how regular datatypes with a kind signature can omit some type variables.
- There are now fewer restrictions regarding whether kind variables can appear on the right-hand sides of type and data family instances. Before, there was a strict requirements that all kind variables on the RHS had to be explicitly bound by type patterns on the LHS. Now, kind variables can be *implicitly* bound, which allows constructions like these:

```
data family Nat :: k -> k -> *
-- k is implicitly bound by an invisible kind pattern
newtype instance Nat :: (k -> *) -> (k -> *) -> * where
  Nat :: (forall xx. f xx -> g xx) -> Nat f g

class Funct f where
```

```

type Codomain f :: *
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 -> *)

```

- Implicitly bidirectional pattern synonyms no longer allow bang patterns (!) or irrefutable patterns (~) on the right-hand side. Previously, this was allowed, although the bang patterns and irrefutable patterns would be silently ignored when used in an expression context. This is now a proper error, and explicitly bidirectional pattern synonyms should be used in their stead. That is, instead of using this (which is an error):

```
data StrictJust a = Just !a
```

Use this:

```

data StrictJust a <- Just !a where
  StrictJust !a = Just a

```

- GADTs with kind-polymorphic type arguments now require *-XTypeInType* (page 352). For instance, consider the following,

```

data G :: k -> * where
  GInt   :: G Int
  GMaybe :: G Maybe

```

In previous releases this would compile with *-XPolyKinds* (page 352) alone due to bug [Trac #13391](#). As of GHC 8.4, however, this requires *-XTypeInType* (page 352). Note that since GADT kind signatures aren't generalized, this will also require that you provide a *CUSK* (page 354) by explicitly quantifying over the kind argument, *k*,

```

data G :: forall k. k -> * where
  GInt   :: G Int
  GMaybe :: G Maybe

```

- The order in which type variables are quantified in GADT constructor type signatures has changed. Before, if you had *MkT* as below:

```

data T a where
  MkT :: forall b a. b -> T a

```

Then the type of *MkT* would (counterintuitively) be *forall a b. b -> T a*! Now, GHC quantifies the type variables in the order that the users writes them, so the type of *MkT* is now *forall b a. b -> T a* (this matters for *-XTypeApplications* (page 375)).

- The new *-XEmptyDataDeriving* (page 286) extension allows deriving *Eq*, *Ord*, *Read*, and *Show* instances directly for empty data types, as in `data Empty deriving Eq`. (Previously, this would require the use of *-XStandaloneDeriving* (page 287) to accomplish.)

One can also now derive *Data* instances directly for empty data types (as in `data Empty deriving Data`) without needing to use *-XStandaloneDeriving* (page 287). However, since already requires a GHC extension (*-XDeriveDataTypeable* (page 295)), one does not need to enable *-XEmptyDataDeriving* (page 286) to do so. This also goes for other classes which require extensions to derive, such as *-XDeriveFunctor* (page 289).

- Hexadecimal floating point literals (e.g. `0x0.1p4`), enabled with *-XHexFloatLiterals* (page 242). See *Hexadecimal floating point literals* (page 242) for the full details.

3.2.2 Compiler

- LLVM code generator (e.g. `-fllvm` (page 186)) compatible with LLVM releases in the 5.0 series.
- Add warning flag `-Wmissing-export-lists` (page 82) which causes the type checker to warn when a module does not include an explicit export list.
- The configure script now no longer accepts `--with-TOOL` flags (e.g. `--with-nm`, `--with-ld`, etc.). Instead, these are taken from environment variables, as is typical in autoconf scripts. For instance, `./configure --with-nm=/usr/local/bin/nm` turns into `./configure NM=/usr/local/bin/nm`.
- Derived Functor, Foldable, and Traversable instances are now optimized when their last type parameters have phantom roles. Specifically,

```
fmap _ = coerce
traverse _ x = pure (coerce x)
foldMap _ _ = mempty
```

These definitions of `foldMap` and `traverse` are lazier than the ones we would otherwise derive, as they may produce results without inspecting their arguments at all.

See also [Deriving Functor instances](#) (page 289), [Deriving Foldable instances](#) (page 292), and [Deriving Traversable instances](#) (page 294).

- Derived instances for empty data types are now substantially different than before. Here is an overview of what has changed. These examples will use a running example of data `Empty a` to describe what happens when an instance is derived for `Empty`:
 - Derived `Eq` and `Ord` instances would previously emit code that used `error`:

```
instance Eq (Empty a) where
  (==) = error "Void =="
```

```
instance Ord (Empty a) where
  compare = error "Void compare"
```

Now, they emit code that uses maximally defined, lazier semantics:

```
instance Eq (Empty a) where
  _ == _ = True
```

```
instance Ord (Empty a) where
  compare _ _ = EQ
```

- Derived `Read` instances would previous emit code that used `parens`:

```
instance Read (Empty a) where
  readPrec = parens pfail
```

But `parens` forces parts of the parsed string that it doesn't need to. Now, the derived instance will not use `parens` (that is, parsing `Empty` will always fail, without reading any input):

```
instance Read (Empty a) where
  readPrec = pfail
```

- Derived `Show` instances would previously emit code that used `error`:

```
instance Show (Empty a) where
  showsPrec = "Void showsPrec"
```

Now, they emit code that inspects the argument. That is, if the argument diverges, then showing it will also diverge:

```
instance Show (Empty a) where
  showsPrec _ x = case x of {}
```

- Derived Functor, Foldable, Traversable, Generic, Generic1, Lift, and Data instances previously emitted code that used error:

```
instance Functor Empty where
  fmap = error "Void fmap"

instance Foldable Empty where
  foldMap = error "Void foldMap"

instance Traversable Empty where
  traverse = error "Void traverse"

instance Generic (Empty a) where
  from = M1 (error "No generic representation for empty datatype Empty")
  to (M1 _) = error "No values for empty datatype Empty"
-- Similarly for Generic1

instance Lift (Empty a) where
  lift _ = error "Can't lift value of empty datatype Empty"

instance Data a => Data (Empty a) where
  gfoldl _ _ _ = error "Void gfoldl"
  toConstr _ = error "Void toConstr"
...
```

Now, derived Functor, Traversable, ``Generic, Generic1, Lift, and Data instances emit code which inspects their arguments:

```
instance Functor Empty where
  fmap _ x = case x of {}

instance Traversable Empty where
  traverse _ x = pure (case x of {})

instance Generic (Empty a) where
  from x = M1 (case x of {})
  to (M1 x) = case x of {}

-- Similarly for Generic1

instance Lift (Empty a) where
  lift x = pure (case x of {})

instance Data a => Data (Empty a) where
  gfoldl _ x = case x of {}
  toConstr x = case x of {}
...
```

Derived Foldable instances now are maximally lazy:

```
instance Foldable Empty where
  foldMap _ _ = mempty
```

- Derived Foldable instances now derive custom definitions for null instead of using the default one. This leads to asymptotically better performance for recursive types not shaped like cons-lists, and allows null to terminate for more (but not all) infinitely large structures.
- Configure on Windows now supports the `--enable-distro-toolchain` configure flag, which can be used to build a GHC using compilers on your PATH instead of using the bundled bindist. See [Trac #13792](#)
- GHC now enables `-fllvm-pass-vectors-in-regs` (page 92) by default. This means that GHC will now use native vector registers to pass vector arguments across function calls.
- The optional instance keyword is now usable in type family instance declarations. See [Trac #13747](#)
- Lots of other bugs. See [Trac](#) for a complete list.
- New flags `-fignore-optim-changes` (page 152) and `-fignore-hpc-changes` (page 152) allow GHC to reuse previously compiled modules even if they were compiled with different optimisation or HPC flags. These options are enabled by default by `--interactive` (page 66). See [Trac #13604](#)

3.2.3 Runtime system

- Function `hs_add_root()` was removed. It was a no-op since GHC-7.2.1 where module initialisation stopped requiring a call to `hs_add_root()`.
- Proper import library support added to GHC which can handle all of the libraries produced by `dlltool`. The limitation of them needing to be named with the suffix `.dll.a` is also removed. See [Trac #13606](#), [Trac #12499](#), [Trac #12498](#)
- The GHCi runtime linker on Windows now supports the `big-obj` file format.
- The runtime system's `native stack backtrace` (page 487) support on POSIX platforms is now triggered by `SIGQUIT` instead of `SIGUSR2` as it was in previous releases. This change is to bring GHC's behavior into compliance with the model set by the most Java virtual machine implementations.
- The GHC runtime on Windows now uses Continue handlers instead of Vectorized handlers to trap exceptions. This change gives other exception handlers a chance to handle the exception before the runtime does. Furthermore The RTS flag `--install-seh-handlers=` Can be used on Windows to completely disable the runtime's handling of exceptions. See [Trac #13911](#), [Trac #12110](#).
- The GHC runtime on Windows can now generate crash dumps on unhandled exceptions using the RTS flag `--generate-crash-dumps` (page 133).
- The GHCi runtime linker now avoid calling GCC to find libraries as much as possible by caching the list of search directories of GCC and querying the file system directly. This results in much better performance, especially on Windows.
- The GHC runtime on Windows can now generate stack traces on unhandled exceptions. When running in GHCi more information is displayed about the symbols if available. This behavior can be controlled with the RTS flag `-generate-stack-traces=<yes|no>`.

3.2.4 Template Haskell

- Template Haskell now reifies data types with GADT syntax accurately. Previously, TH used heuristics to determine whether a data type should be reified using GADT syntax, which could lead to incorrect results, such as `data T1 a = (a ~ Int) => MkT1` being reified as a GADT and `data T2 a where MkT2 :: Show a => T2 a` *not* being reified as a GADT.

In addition, reified GADT constructors now more accurately track the order in which users write type variables. Before, if you reified `MkT` as below:

```
data T a where
  MkT :: forall b a. b -> T a
```

Then the reified type signature of `MkT` would have been headed by `ForallC [PlainTV a, PlainTV b]`. Now, reifying `MkT` will give a type headed by `ForallC [PlainTV b, PlainTV a]`, as one would expect.

- `Language.Haskell.TH.FamFlavour`, which was deprecated in GHC 8.2, has been removed.

3.2.5 ghc library

- `hsSyn` Abstract Syntax Tree (AST) is now extensible via the mechanism described in [Trees that Grow](#)

The main change for users of the GHC API is that the AST is no longer indexed by the type used as the identifier, but by a specific index type,

```
type GhcPs    = GhcPass 'Parsed      -- Old 'RdrName' type param
type GhcRn    = GhcPass 'Renamed    -- Old 'Name' type param
type GhcTc    = GhcPass 'Typechecked -- Old 'Id' type para,
type GhcTcId  = GhcTc               -- Old 'TcId' type param
```

The simplest way to support the current GHC as well as earlier ones is to define

```
#if MIN_VERSION_ghc(8,3,0)
type ParseI    = GhcPs
type RenameI    = GhcRn
type TypecheckI = GhcTc
#else
type ParseI    = RdrName
type RenameI    = Name
type TypecheckI = Var
#endif
```

and then replace all hardcoded index types accordingly. For polymorphic types, the constraint

```
#if MIN_VERSION_ghc(8,3,0)
-- |bundle up the constraints required for a trees that grow pass
type IsPass pass = (DataId pass, OutputableBndrId pass, SourceTextX pass)
else
type IsPass pass = (DataId pass, OutputableBndrId pass)
#endif
```

can be used.

3.2.6 base library

- Blank strings can now be used as values for environment variables using the `System.Environment.Blank` module. See [Trac #12494](#)
- `Data.Type.Equality.==` is now a closed type family. It works for all kinds out of the box. Any modules that previously declared instances of this family will need to remove them. Whereas the previous definition was somewhat ad hoc, the behavior is now completely uniform. As a result, some applications that used to reduce no longer do, and conversely. Most notably, `(==)` no longer treats the `*`, `j -> k`, or `()` kinds specially; equality is tested structurally in all cases.

3.2.7 Build system

- `dll-split` has been removed and replaced with an automatic partitioning utility `gendll`. This utility can transparently split and compile any DLLs that require this. Note that the `rts` and `base` can not be split at this point because of the mutual recursion between `base` and `rts`. There is currently no explicit dependency between the two in the build system and such there is no way to notify `base` that the `rts` has been split, or vice versa. (see [Trac #5987](#)).

3.3 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	8.4.0.20180118	The compiler itself
Cabal	2.1.0.0	Dependency of ghc-pkg utility
Win32	2.6.1.0	Dependency of ghc library
array	0.5.2.0	Dependency of ghc library
base	2.1	Core library
binary	0.8.5.1	Dependency of ghc library
bytestring	0.10.8.2	Dependency of ghc library
containers	0.5.10.2	Dependency of ghc library
deepseq	1.4.3.0	Dependency of ghc library
directory	1.3.1.5	Dependency of ghc library
filepath	1.4.1.2	Dependency of ghc library
ghc-boot	8.4.0.20180118	Internal compiler library
ghc-compact	0.1.0.0	Core library
ghc-prim	2.1	Core library
ghci	8.4.0.20180118	The REPL interface
haskeline	0.7.4.0	Dependency of ghci executable
hpc	0.6.0.3	Dependency of hpc executable
integer-gmp	1.0.1.0	Core library
mtl	2.2.2	Dependency of Cabal library
parsec	3.1.12	Dependency of Cabal library
process	1.6.1.0	Dependency of ghc library
template-haskell	2.13.0.0	Core library

Continued on next page

Table 3.1 – continued from previous page

Package	Version	Reason for inclusion
text	1.2.3.0	Dependency of Cabal library
time	1.8.0.2	Dependency of ghc library
transformers	0.5.4.0	Dependency of ghc library
unix	2.7.2.2	Dependency of ghc library
xhtml	3000.2.2	Dependency of haddock executable

3.3.1 Win32

Attention: This release is a backwards incompatible release which corrects the type of certain APIs. See issue [#24](#).

USING GHCi

GHCi ¹ is GHC's interactive environment, in which Haskell expressions can be interactively evaluated and programs can be interpreted. If you're familiar with [Hugs](#), then you'll be right at home with GHCi. However, GHCi also has support for interactively loading compiled code, as well as supporting all ² the language extensions that GHC provides. GHCi also includes an interactive debugger (see [The GHCi Debugger](#) (page 33)).

4.1 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: http://www.haskell.org/ghc/  :? for help
Prelude>
```

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 47) 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:

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

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:

```
Prelude> x = 42
Prelude> x
42
Prelude>
```

¹ The "i" stands for "Interactive"

² except `foreign export`, at the moment

4.2 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 directory³ then we will need to change to the right directory in GHCi:

```
Prelude> :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 48) command:

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

GHCi has loaded the `Main` module, and the prompt has changed to `*Main>` 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 26)). So we can now type expressions involving the functions from `Main.hs`:

```
*Main> fac 17
355687428096000
```

Loading a multi-module program is just as straightforward; just give the name of the “topmost” module to the `:load` (page 48) command (hint: `:load` (page 48) 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 48) 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.

4.2.1 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 48), or specify it when you invoke `ghci`, you can give a filename rather than a module name. This

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`.

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 148) option on the GHCi command line, like so:

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

or it can be set using the `:set` (page 50) command from within GHCi (see [Setting GHC command-line options in GHCi](#) (page 55))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.

4.2.2 Making changes and recompilation

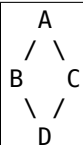
If you make some changes to the source code and want GHCi to recompile the program, give the `:reload` (page 50) command. The program will be recompiled as necessary, with GHCi doing its best to avoid actually recompiling modules if their external dependencies haven't changed. This is the same mechanism we use to avoid re-compiling modules in the batch compilation setting (see [The recompilation checker](#) (page 152)).

4.3 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 48), 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`:



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

⁴ Note that in GHCi, and `--make` (page 66) mode, the `-i` (page 148) option is used to specify the search path for *source* files, whereas in standard batch-compilation mode the `-i` (page 148) option is used to specify the search path for interface files, see [The search path](#) (page 148).

```
Prelude> :! ghc -c -dynamic D.hs
Prelude> :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).
*Main>
```

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 188) 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 52) to get a list of the modules currently loaded into GHCi:

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

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:

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

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:

```
*Main> :! ghc -c C.hs
*Main> :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:

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

Nothing happened! Here's another lesson: newly compiled modules aren't picked up by *:reload* (page 50), only *:load* (page 48):

```
*Main> :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 26)). 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 48), for example

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

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 186) option (see [Compiling to object code inside GHCi](#) (page 57)).

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.

4.4 Interactive evaluation at the prompt

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

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

4.4.1 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.

```
Prelude> "hello"
"hello"
Prelude> 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

```
Prelude> 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 ()`:

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

4.4.2 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.

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

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 22) 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:

```
Prelude> let x = 42
Prelude> x
42
Prelude>
```

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:

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

Note that `let` bindings do not automatically print the value bound, unlike monadic bindings. You can also define functions at the prompt:

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

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:

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

```
:{
:}
```

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):

```
Prelude> :{
Prelude| g op n [] = n
Prelude| g op n (h:t) = h `op` g op n t
Prelude| :}
Prelude> 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 56)) 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 48) or `:reload` (page 50) command, at which time they will be simply lost. However, they do survive a change of context with `:module` (page 49): 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 52) command:

```
Prelude> :show bindings
x :: Int
Prelude>
```

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

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

4.4.3 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:

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

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:

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

Explicit braces and semicolons can be used instead of layout:

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

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:

```
Control.Monad.State> flip evalStateT 0 $ do
Control.Monad.State| i <- get
Control.Monad.State| lift $ do
Control.Monad.State|   putStrLn "Hello World!"
Control.Monad.State|   print i
Control.Monad.State|
"Hello World!"
0
Control.Monad.State>
```

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

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

4.4.4 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:

```
Prelude> data T = A | B | C deriving (Eq, Ord, Show, Enum)
Prelude> [A ..]
[A,B,C]
Prelude> :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:

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

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 334).) For example:

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

Prelude> 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.

Prelude> type family T a b
-- This is a brand-new T, unrelated to the old one
```

```
Prelude> type instance T a b = b
Prelude> 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
```

4.4.5 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 48), `:add` (page 44), and `:reload` (page 50) commands (*The effect of :load on what is in scope* (page 26)).
- The import declaration (*Controlling what is in scope with import* (page 27)).
- The `:module` (page 49) command (*Controlling what is in scope with the :module command* (page 28)).

The command `:show imports` (page 52) 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`.

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

The `:load` (page 48), `:add` (page 44), and `:reload` (page 50) commands (*Loading source files* (page 18) and *Loading compiled code* (page 19)) affect the top-level scope. Let's start with the simple cases; when you start GHCi the prompt looks like this:

```
Prelude>
```

which indicates that everything from the module `Prelude` is currently in scope; the visible identifiers are exactly those that would be visible in a Haskell source file with no `import` declarations.

If we now load a file into GHCi, the prompt will change:

```
Prelude> :load Main.hs
Compiling Main          ( Main.hs, interpreted )
*Main>
```

The new prompt is `*Main`, which indicates that we are typing expressions in the context of the top-level of the `Main` module. Everything that is in scope at the top-level in the module `Main` we just loaded is also in scope at the prompt (probably including `Prelude`, as long as `Main` doesn't explicitly hide it).

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 48) 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 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 52):

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

and the automatically-added import is replaced the next time you use `:load` (page 48), `:add` (page 44), or `:reload` (page 50). It can also be removed by `:module` (page 49) as with normal imports.

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:

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

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 52):

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

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

Another way to manipulate the scope is to use the `:module` (page 49) 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 49) command provides a way to do two things that cannot be done with ordinary import declarations:

- `:module` (page 49) 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.

Qualified names

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.

`:module` and `:load`

It might seem that `:module` (page 49)/`import` and `:load` (page 48)/`:add` (page 44)/`:reload` (page 50) 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 48), `:add` (page 44) and `:reload` (page 50), and can be shown with `:show modules` (page 52).
- The set of modules that are currently *in scope* at the prompt. This set is modified by `import` and `:module` (page 49), and it is also modified automatically after `:load` (page 48), `:add` (page 44), and `:reload` (page 50), as described above. The set of modules in scope can be shown with `:show imports` (page 52).

You can add a module to the scope (via `:module` (page 49) 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 49) or `import` to try bring into scope a non-loaded module may result in the message `module M is not loaded`.

4.4.6 The `:main` and `:run` commands

When a program is compiled and executed, it can use the `getArgs` function to access the command-line arguments. However, we cannot simply pass the arguments to the `main` function while we are testing in `ghci`, as the `main` function doesn't take its directly.

Instead, we can use the `:main` (page 49) command. This runs whatever `main` is in scope, with any arguments being treated the same as command-line arguments, e.g.:

```
Prelude> main = System.Environment.getArgs >>= print
Prelude> :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:

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

Finally, other functions can be called, either with the `-main-is` flag or the `:run` (page 50) command:

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

4.4.7 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:

```
Prelude> 1+2
3
Prelude> 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:

```
Prelude> 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.:

```
Prelude> Data.Time.getZonedTime
2017-04-10 12:34:56.93213581 UTC
Prelude> 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.

4.4.8 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 30) 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 ()

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 21) for how the monad of a computational expression defaults to `IO` if possible.

Interactive classes

The interactive classes (only relevant when *ExtendedDefaultRules* (page 30) 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.

Extended rules around default declarations

Since the rules for defaulting are relaxed under *ExtendedDefaultRules* (page 30), 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)
```

4.4.9 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 (expr)` (page 32) 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), `:load` (page 48), `:reload` (page 50) or `:set` (page 50).

-interactive-print (expr)

Set the function used by GHCi to print evaluation results. Expression 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 will 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 (expr)` (page 32) flag can also be used when running GHC in `-e` mode:

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

4.4.10 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 58)) 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 446)), 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 205)).

4.5 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)).

4.5.1 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])
```

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

First, load the module into GHCi:

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

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

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

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:

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

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 48) command:

```
[qsort.hs:2:15-46] *Main> :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 48) 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 53), 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] *Main> 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 50), 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] *Main> :set -fprint-evld-with-show
[qsort.hs:2:15-46] *Main> :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 50) does not force any evaluation. The idea is that `:print` (page 50) 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 38)). Rather than forcing thunks, `:print` (page 50) binds each thunk to a fresh variable beginning with an underscore, in this case `_t1`.

The flag `-fprint-evld-with-show` instructs `:print` (page 50) 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 47) instead of `:print` (page 50). The `:force` (page 47) command behaves exactly like `:print` (page 50), except that it forces the evaluation of any thunks it encounters:

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

Now, since `:force` (page 47) 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] *Main> :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] *Main> 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 47). For example:

```
[qsort.hs:2:15-46] *Main> :print right
right = (_t1::[Integer])
[qsort.hs:2:15-46] *Main> seq _t1 ()
()
[qsort.hs:2:15-46] *Main> :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] *Main> :continue
Stopped at qsort.hs:2:15-46
_result :: [a]
a :: a
left :: [a]
right :: [a]
[qsort.hs:2:15-46] *Main>
```

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

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 but before any pattern matching has taken place.

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 84) in [Warnings and sanity-checking](#) (page 74)).

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.

Listing and deleting breakpoints

The list of breakpoints currently enabled can be displayed using `:show breaks` (page 52):

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

To delete a breakpoint, use the `:delete` (page 47) command with the number given in the output from `:show breaks` (page 52):

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

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

4.5.2 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 52) to enable all the breakpoints in the program, and execute until the next breakpoint is reached. Use `:steplocal` (page 52) to limit the set of enabled breakpoints to those in the current top level function. Similarly, use `:stepmodule` (page 52) to single step only on breakpoints contained in the current module. For example:

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

The command `:step expr` (page 52) begins the evaluation of `(expr)` in single-stepping mode. If `(expr)` is omitted, then it single-steps from the current breakpoint. `:steplocal` (page 52) and `:stepmodule` (page 52) commands work similarly.

The `:list` (page 48) command is particularly useful when single-stepping, to see where you currently are:

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

In fact, GHCi provides a way to run a command when a breakpoint is hit, so we can make it automatically do `:list` (page 48):

```
[qsort.hs:5:7-47] *Main> :set stop :list
[qsort.hs:5:7-47] *Main> :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] *Main>
```

4.5.3 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] *Main> :st qsort [1,3]
Stopped at qsort.hs:(1,0)-(3,55)
_result :: [a]
... [qsort.hs:(1,0)-(3,55)] *Main>
```

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 52):

```
... [qsort.hs:(1,0)-(3,55)] *Main> :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)] *Main>
```

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

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

4.5.4 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 52)). So it will probably be necessary to issue a `:continue` (page 46) immediately when evaluating `_result`. Alternatively, you can use `:force` (page 47) which ignores breakpoints.

4.5.5 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 203)), 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 52) command. For example, if we set a breakpoint on the base case of `qsort`:

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

and then run a small `qsort` with tracing:

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

We can now inspect the history of evaluation steps:

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

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 47) can be used to traverse forward in the history.

The `:trace` (page 52) command can be used with or without an expression. When used without an expression, tracing begins from the current breakpoint, just like `:step` (page 52).

The history is only available when using `:trace` (page 52); 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.

4.5.6 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 145)).

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 52) and `:history` (page 47) 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 41), 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 52), in order to log the steps leading up to the exception. For example:

```
*Main> :set -fbreak-on-exception
*Main> :trace qsort ("abc" ++ undefined)
"Stopped at <exception thrown>
_exception :: e
[<exception thrown>] *Main> :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>] *Main> :back
Logged breakpoint at qsort.hs:3:24-38
_result :: [a]
as :: [a]
a :: a
[-1: qsort.hs:3:24-38] *Main> :force as
*** Exception: Prelude.undefined
[-1: qsort.hs:3:24-38] *Main> :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 41) breaks on only those exceptions which would otherwise be uncaught.

4.5.7 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.

```
*Main> :break 5
Breakpoint 0 activated at map.hs:5:15-28
*Main> 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 33)), 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`.

```
*Main> seq x ()
*Main> :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:

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

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

```
*Main> let b = f 10
*Main> :t b
b :: b
*Main> b
<interactive>:1:0:
  Ambiguous type variable `b' in the constraint:
    `Show b' arising from a use of `print' at <interactive>:1:0
*Main> :p b
b = (_t2::a)
*Main> seq b ()
()
*Main> :t b
b :: a
*Main> :p b
b = Just 10
*Main> :t b
b :: Maybe Integer
*Main> :t f
f :: Integer -> Maybe Integer
*Main> f 20
Just 20
*Main> 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.

4.5.8 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 377)) are only available at the scope of a breakpoint if there is an explicit type signature.

4.6 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 63)) 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 `~/.ghc/ghci_history` or `%APP-DATA%/<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.

4.6.1 Packages

Most packages (see [Using Packages](#) (page 164)) 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 165) flag:

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

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

```
Prelude> :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.

4.6.2 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 18).) For example, to load the “m” library:

```
$ ghci -lm
```

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

- Paths specified using the `-L (dir)` (page 188) command-line option,
- the standard library search path for your system, which on some systems may be overridden by setting the `LD_LIBRARY_PATH` environment variable.

On systems with `.dll`-style shared libraries, the actual library loaded will be `lib.dll`. 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) from the command-line. Just add the name the object file to the command line.

Ordering of `-l` options matters: a library should be mentioned *before* the libraries it depends on (see [Options affecting linking](#) (page 187)).

4.7 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 54) 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 52), *:history* (page 47), *:forward* (page 47).

:break [(identifier) | [(module)] (line) [(column)]]

Set a breakpoint on the specified function or line and column. See *Setting breakpoints* (page 36).

: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 26)).

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 than 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:

```

Prelude> :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.toList :: [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 26)), 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 52) 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 `I0 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 51).

: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 $\langle n \rangle^{\text{th}}$ to $\langle m \rangle^{\text{th}}$ 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, $\langle n \rangle$ and $\langle m \rangle$ default to the first or last available completion candidate respectively. If there are less candidates than requested via the range argument, $\langle n \rangle$ and $\langle m \rangle$ 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:

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

:continue

Continue the current evaluation, when stopped at a breakpoint.

:ctags [{filename}]

Generates a “tags” file for Vi-style editors ([:ctags](#) (page 46)) or Emacs-style editors ([:etags](#) (page 47)). If no filename is specified, the default tags or TAGS is used, respectively. Tags for all the functions, constructors and types in the currently loaded modules are created. All modules must be interpreted for these commands to work.

: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:

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

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

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

```
Prelude> :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:

```
Prelude> :def . readFile
Prelude> :. 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.

`:delete` * | (num) ...

Delete one or more breakpoints by number (use `:show breaks` (page 52) to see the number of each breakpoint). The `*` form deletes all the breakpoints.

`: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 EDITOR environment variable, or a default editor on your system if EDITOR is not set. You can change the editor using `:set editor` (page 50).

`:etags`

See `:ctags` (page 46).

`:force` (identifier) ...

Prints the value of (identifier) in the same way as `:print` (page 50). Unlike `:print` (page 50), `:force` (page 47) 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 52), `:history` (page 47), `: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 52); see *Tracing and history* (page 38). To set the number of history entries stored by GHCi, use the `-fghci-hist-size={n}` (page 40) 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 48) or `:module` (page 49) 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.

: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.

: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 48) 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 :: * -> *
```

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 48) 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” 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 393) for further motivation and details.

After a `:load` (page 48) 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 48), 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 54) to be set.

:main (arg1) ... (argn)

When a program is compiled and executed, it can use the `getArgs` function to access the command-line arguments. However, we cannot simply pass the arguments to the `main` function while we are testing in `ghci`, as the `main` function doesn't take its arguments directly.

Instead, we can use the `:main` (page 49) command. This runs whatever `main` is in scope, with any arguments being treated the same as command-line arguments, e.g.:

```
Prelude> main = System.Environment.getArgs >>= print
Prelude> :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:

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

Finally, other functions can be called, either with the `-main-is` flag or the `:run` (page 50) command:

```
Prelude> foo = putStrLn "foo" >> System.Environment.getArgs >>= print
Prelude> bar = putStrLn "bar" >> System.Environment.getArgs >>= print
Prelude> :set -main-is foo
Prelude> :main foo "bar baz"
foo
["foo","bar baz"]
Prelude> :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 26) for more details.

:print {names}

Prints a value without forcing its evaluation. `:print` (page 50) 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 50) 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 50) binds a fresh variable with a name beginning with `_t` to each thunk. See *Breakpoints and inspecting variables* (page 33) for more information. See also the `:sprint` (page 52) command, which works like `:print` (page 50) 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 48)) 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` 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 393) for further motivation and details.

:run

See `:main` (page 49).

:script [{n}] {filename}

Executes the lines of a file as a series of GHCi commands. This command is compatible with multiline statements as set by `:set +m` (page 54)

:set [{option} ...]

Sets various options. See *The :set and :seti commands* (page 54) for a list of available options and *Interactive-mode options* (page 106) for a list of GHCi-specific flags. The `:set` (page 50) 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.getArgs`.

:set editor {cmd}

Sets the command used by `:edit` (page 47) to {cmd}.

:set prog {prog}

Sets the string to be returned when the program calls `System.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 23) 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 26) for more information). The second argument 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 23) 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 51) 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 (although GHCi will still emit a message to say the breakpoint was hit). What's more, with cunning use of `:def` (page 46) and `:cmd` (page 45) you can use `:set stop` (page 51) to implement conditional breakpoints:

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

Ignoring breakpoints for a specified number of iterations is also possible using similar techniques.

:seti [(option) ...]

Like `:set` (page 50), but options set with `:seti` (page 51) affect only expressions and

commands typed at the prompt, and not modules loaded with `:load` (page 48) (in contrast, options set with `:set` (page 50) apply everywhere). See *Setting options for interactive evaluation only* (page 55).

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 49) 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 51)).

`:show [args|prog|prompt|editor|stop]`

Displays the specified setting (see `:set` (page 50)).

`:sprint (expr)`

Prints a value without forcing its evaluation. `:sprint` (page 52) is similar to `:print` (page 50), 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.

`: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 47). See [Tracing and history](#) (page 38).

:type (expression)

Infers and prints the type of (expression), including explicit forall quantifiers for polymorphic types. The type reported is the type that would be inferred for a variable assigned to the expression, but without the monomorphism restriction applied.

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

:type +v (expression)

Infers and prints the type of (expression), but without fiddling with type variables or class constraints. This is useful when you are using [TypeApplications](#) (page 375) and care about the distinction between specified type variables (available for type application) and inferred type variables (not available). This mode sometimes prints constraints (such as `Show Int`) that could readily be solved, but solving these constraints may affect the type variables, so GHC refrains.

```
*X> :set -fprint-explicit-foralls
*X> :type +v length
length :: forall (t :: * -> *). Foldable t => forall a. t a -> Int
```

:type +d (expression)

Infers and prints the type of (expression), defaulting type variables if possible. 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 30). 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 (module) (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 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 53) falls back to a general `:type` (page 53) like lookup.

The `:type-at` (page 53) command requires `:set +c` (page 54) to be set.

: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 54) 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)
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 53) command requires `:set +c` (page 54) to be set.

:! (command)

Executes the shell command (command).

4.8 The `:set` and `:seti` commands

The `:set` (page 50) 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 50) 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 -DF00='BAR BAZ'` will not do what you expect.

4.8.1 GHCi options

GHCi options may be set using `:set` (page 50) and unset using `:unset` (page 53).

The available GHCi options are:

:set +c

Collect type and location information after loading modules. The commands `:all-types` (page 44), `:loc-at` (page 49), `:type-at` (page 53), and `:uses` (page 53) 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 24)).

: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`.

4.8.2 Setting GHC command-line options in GHCi

Normal GHC command-line options may also be set using `:set` (page 50). For example, to turn on `-Wmissing-signatures` (page 82), you would say:

```
Prelude> :set -Wmissing-signatures
```

Any GHC command-line option that is designated as dynamic (see the table in [Flag reference](#) (page 101)), may be set using `:set` (page 50). To unset an option, you can set the reverse option:

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

[Flag reference](#) (page 101) lists the reverse for each option where applicable.

Certain static options (`-package <pkg>` (page 165), `-I<dir>` (page 183), `-i<dir>[:<dir>]*` (page 148), and `-l <lib>` (page 187) in particular) will also work, but some may not take effect until the next reload.

4.8.3 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 50) command modifies both, but there is also a `:seti` (page 51) 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 374) 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 51) rather than `:set` (page 50), 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 50) and `:seti` (page 51) commands respectively, with no arguments. For example, in a clean GHCi session we might see something like this:

```
Prelude> :seti
base language is: Haskell2010
with the following modifiers:
  -XNoMonomorphismRestriction
  -XNoDatatypeContexts
  -XNondecreasingIndentation
  -XExtendedDefaultRules
```

GHCi-specific dynamic flag settings:
other dynamic, non-language, flag settings:
 -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 56). 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 30)).
- The Monomorphism Restriction is disabled (see *Switching off the dreaded Monomorphism Restriction* (page 374)).

4.9 The .ghci and .haskeline files

4.9.1 The .ghci files

When it starts, unless the `-ignore-dot-ghci` (page 57) flag is given, GHCi reads and executes commands from the following files, in this order, if they exist:

1. `./ .ghci`
2. `appdata/ghc/ghci.conf`, where (appdata) depends on your system, but is usually something like `C:/Documents and Settings/user/Application Data`
3. On Unix: `$HOME/.ghc/ghci.conf`
4. `$HOME/.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 51) rather than `:set` (page 50) (see *Setting options for interactive evaluation only* (page 55)).

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 148) flag is a static one, but in fact it works to set it using `:set` (page 50) like this. The changes won't take effect until the next `:load` (page 48), though.)

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 57) flags will be read after the standard files, allowing the use of custom `.ghci` files.

Two command-line options control whether the startup 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. Maybe be specified repeatedly for multiple inputs.

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.

4.9.2 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](#).

4.10 Compiling to object code inside GHCi

By default, GHCi compiles Haskell source code into byte-code that is interpreted by the run-time system. GHCi can also compile Haskell code to object code: to turn on this feature, use the `-fobject-code` (page 186) flag either on the command line or with `:set` (page 50) (the option `-fbyte-code` (page 186) 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 50) command typically runs much faster than restarting GHC with `--make` (page 66) 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.

4.11 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 58) 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 207) is in effect, and in dynamically-linked mode if `-dynamic` (page 188) 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 58).

See also the `-pgmi <cmd>` (page 182) (*Replacing the program for one or more phases* (page 181)) and `-opti <option>` (page 183) (*Forcing options to a particular phase* (page 182)) 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 32)).
- When compiling Template Haskell code with `-prof` (page 207) we don't need to compile the modules without `-prof` (page 207) first (see *Using Template Haskell with Profiling* (page 401)) 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.

4.12 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 88) doesn't work with GHCi!

For technical reasons, the bytecode compiler doesn't interact well with one of the optimisation passes, so we have disabled optimisation when using the interpreter. This isn't a great loss: you'll get a much bigger win by compiling the bits of your code that need to go fast, rather than interpreting them with optimisation turned on.

Unboxed tuples don't work with GHCi That's right. You can always compile a module that uses unboxed tuples and load it into GHCi, however. (Incidentally the previous point,

namely that `-O` (page 88) is incompatible with GHCi, is because the bytecode compiler can't deal with unboxed tuples).

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 189) 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 48) or `:reload` (page 50) 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 495).

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; ... }
```


USING RUNGHC

runghc allows you to run Haskell programs without first having to compile them.

5.1 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`

5.2 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.

5.3 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`

USING GHC

6.1 Using GHC

6.1.1 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 69) 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.

6.1.2 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:

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`.

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 95) and `-O1` (page 88) (which implies `-fspecialise` (page 95)). 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`.

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 421)

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

`OPTIONS_GHC` is a *file-header pragma* (see *OPTIONS_GHC pragma* (page 421)).

Only *dynamic* flags can be used in an `OPTIONS_GHC` pragma (see *Static, Dynamic, and Mode options* (page 65)).

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 150) and have `OPTION` flags in your module, the `OPTIONS_GHC` will get put into the generated `.hc` file).

Setting options in GHCi

Options may also be modified from within GHCi, using the `:set` (page 50) command.

6.1.3 Static, Dynamic, and Mode options

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

For example, `--make` (page 66) or `-E` (page 66). There may only be a single mode flag on the command line. The available modes are listed in *Modes of operation* (page 66).

Most non-mode flags fall into this category. A dynamic flag may be used on the command line, in a `OPTIONS_GHC` pragma in a source file, or set using `:set` (page 50) in GHCi.

A few flags are “static”, which means they can only be used on the command-line, and remain in force over the entire GHC/GHCi run.

The flag reference tables (*Flag reference* (page 101)) lists the status of each flag.

There are a few flags that are static except that they can also be used with GHCi's `:set` (page 50) command; these are listed as “static/:set” in the table.

6.1.4 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.
- .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.

6.1.5 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 66) mode (*Using ghc -make* (page 67)) 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 17).

--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 67).

This mode is the default if there are any Haskell source files mentioned on the command line, and in this case the `--make` (page 66) 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. See *Expression evaluation mode* (page 68) 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 68).

-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 161).

--frontend {module}

Run GHC using the given frontend plugin. See *Frontend plugins* (page 480) for details.

--mk-dll

DLL-creation mode (Windows only). See *Creating a DLL* (page 498).

--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-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 163)).

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 Makefiles 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 68) 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 64)).

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 66), 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 148) option can be used to add directories to the search path (see [The search path](#) (page 148)).

-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.

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
```

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 180) for more details.

Note: Pre-processing is optional, the `-cpp` (page 183) flag turns it on. See [Options affecting the C pre-processor](#) (page 183) for more details.

Note: The option `-E` (page 66) runs just the pre-processing passes of the compiler, dumping the result in a file.

Note: The option `-C` (page 66) is only available when GHC is built in unregistered mode. See [Unregistered compilation](#) (page 181) 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 69) 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`.

6.1.6 Verbosity options

See also the `--help`, `--version`, `--numeric-version`, and `--print-libdir` modes in [Modes of operation](#) (page 66).

`-v`

The `-v` (page 69) 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 66) or `--interactive` (page 66) is on).
- v2** Print the name of each compilation phase as it is executed. (equivalent to `-dshow-passes` (page 196)).
- 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 in 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 69)), 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 `XUnicodeSyntax` (page 240) 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 70) 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
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 :: BOX) (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 71) 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 = MkT
ghci> :t MkT
MkT :: forall (k :: BOX) (a :: k). T a
ghci> :set -fprint-explicit-foralls
ghci> :t MkT
MkT :: forall (k :: BOX) (a :: k). T k a
```

-fprint-explicit-runtime-reps

When *-fprint-explicit-runtime-reps* (page 71) is enabled, GHC prints RuntimeRep type variables for levity-polymorphic types. Otherwise GHC will default these to PtrRepLifted. 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
```

-fprint-explicit-coercions

Using *-fprint-explicit-coercions* (page 71) 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-equality-relations

Using *-fprint-equality-relations* (page 71) 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 71), GHC prints all of these as `~`. See also *Equality constraints* (page 364).

-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-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
               <>]
     Exported types: a :: [Char]
                    [LclId, Str=DmdType]
     Binds: a = "hello"}
  in a'
```

or by using the flag `-fno-warn-unused-do-bind`

-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

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. The flag is on by default.

-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).

-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 134).

-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 134).

6.1.7 Platform-specific Flags

Some flags only make sense for particular target platforms.

-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 180). 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 180) will also use SSE2 if your processor supports it but detects this automatically so no flag is required.

SSE2 is unconditionally used on x86-64 platforms.

-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 180). The resulting compiled code will only run on processors that support SSE4.2 (Intel Core i7 and later). The *LLVM backend* (page 180) will also use SSE4.2 if your processor supports it but detects this automatically so no flag is required.

6.1.8 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.

6.2 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. By default, you get a standard set of warnings which are generally likely to indicate bugs in your program. These are:

- *-Woverlapping-patterns* (page 83)
- *-Wwarnings-deprecations* (page 77)
- *-Wdeprecations* (page 78)
- *-Wdeprecated-flags* (page 79)
- *-Wunrecognised-pragmas* (page 77)
- *-Wduplicate-constraints* (page 80)
- *-Wduplicate-exports* (page 80)
- *-Woverflowed-literals* (page 79)
- *-Wempty-enumerations* (page 79)
- *-Wmissing-fields* (page 82)
- *-Wmissing-methods* (page 82)
- *-Wwrong-do-bind* (page 86)
- *-Wunsupported-calling-conventions* (page 79)
- *-Wdodgy-foreign-imports* (page 79)
- *-Winline-rule-shadowing* (page 87)

- *-Wunsupported-llvm-version* (page 84)
- *-Wtabs* (page 84)
- *-Wunrecognised-warning-flags* (page 76)

The following flags are simple ways to select standard “packages” of warnings:

-W

Provides the standard warnings plus

- *-Wunused-binds* (page 84)
- *-Wunused-matches* (page 85)
- *-Wunused-foralls* (page 86)
- *-Wunused-imports* (page 85)
- *-Wincomplete-patterns* (page 81)
- *-Wdodgy-exports* (page 79)
- *-Wdodgy-imports* (page 79)
- *-Wunbanged-strict-patterns* (page 87)

-Wall

Turns on all warning options that indicate potentially suspicious code. The warnings that are *not* enabled by *-Wall* (page 75) are

- *-Wincomplete-uni-patterns* (page 81)
- *-Wincomplete-record-updates* (page 81)
- *-Wmonomorphism-restriction* (page 84)
- *-Wimplicit-prelude* (page 81)
- *-Wmissing-local-signatures* (page 83)
- *-Wmissing-exported-signatures* (page 83)
- *-Wmissing-export-lists* (page 82)
- *-Wmissing-import-lists* (page 82)
- *-Wmissing-home-modules* (page 87)
- *-Widentities* (page 80)
- *-Wredundant-constraints* (page 80)
- *-Wpartial-fields* (page 87)

-Weverything

Turns on every single warning supported by the compiler.

-Wcompat

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

- *-Wmissing-monadfail-instances* (page 78)
- *-Wsemigroup* (page 79)
- *-Wnoncanonical-monoid-instances* (page 78)

-Wno-compat

Disables all warnings enabled by *-Wcompat* (page 75).

-w

Turns off all warnings, including the standard ones and those that *-Wall* (page 75) doesn't enable.

These options control which warnings are considered fatal and cause compilation to abort.

-Werror

Makes any warning into a fatal error. Useful so that you don't miss warnings when doing

batch compilation.

-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.

-Wwarn

Warnings are treated only as warnings, not as errors. This is the default, but can be useful to negate a [-Werror](#) (page 75) 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.

When a warning is emitted, the specific warning flag which controls it is shown.

-fshow-warning-groups

When showing which flag controls a warning, also show the respective warning group flag(s) that warning is contained in.

This option is off by default.

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

Enables warnings when the compiler encounters a **-W.** . . flag that is not recognised.

This warning is on by default.

-Wtyped-holes

Determines whether the compiler reports typed holes warnings. Has no effect unless typed holes errors are deferred until runtime. See [Typed Holes](#) (page 385) and [Deferring type errors to runtime](#) (page 393)

This warning is on by default.

-Wdeferred-type-errors

Causes a warning to be reported when a type error is deferred until runtime. See [Deferring type errors to runtime](#) (page 393)

This warning is on by default.

-fdefer-type-errors

Implies [-fdefer-typed-holes](#) (page 76)

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 [Deferring type errors to runtime](#) (page 393)

-fdefer-typed-holes

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 385) into warnings. Using a value that depends on a typed hole produces a runtime error, the

same as *-fdefer-type-errors* (page 76) (which implies this option). See *Typed Holes* (page 385) and *Deferring type errors to runtime* (page 393).

Implied by *-fdefer-type-errors* (page 76). See also *-Wtyped-holes* (page 76).

-fdefer-out-of-scope-variables

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 a typed hole produces a runtime error, the same as *-fdefer-type-errors* (page 76) (which implies this option). See *Typed Holes* (page 385) and *Deferring type errors to runtime* (page 393).

Implied by *-fdefer-type-errors* (page 76). See also *-Wdeferred-out-of-scope-variables* (page 77).

-Wdeferred-out-of-scope-variables

Warn when a deferred out-of-scope variable is encountered.

-Wpartial-type-signatures

Determines whether the compiler reports holes in partial type signatures as warnings. Has no effect unless *-XPartialTypeSignatures* (page 387) is enabled, which controls whether errors should be generated for holes in types or not. See *Partial Type Signatures* (page 387).

This warning is on by default.

-fhelptful-errors

When a name or package is not found in scope, make suggestions for the name or package you might have meant instead.

This option is on by default.

-Wunrecognised-pragmas

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`.

This option is on by default.

-Wmissed-specialisations

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 75).

This option is off by default.

-Wall-missed-specialisations

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 75).

This option is off by default.

-Wwarnings-deprecations

Causes a warning to be emitted when a module, function or type with a `WARNING` or

DEPRECATED pragma is used. See [WARNING and DEPRECATED pragmas](#) (page 421) for more details on the pragmas.

This option is on by default.

-Wdeprecations

Causes a warning to be emitted when a module, function or type with a WARNING or DEPRECATED pragma is used. See [WARNING and DEPRECATED pragmas](#) (page 421) for more details on the pragmas. An alias for [-Wwarnings-deprecations](#) (page 77).

This option is on by default.

-Wwamp

This option is deprecated.

Caused a warning to be emitted when a definition was in conflict with the AMP (Applicative-Monad proposal).

-Wnoncanonical-monad-instances

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. `(*>) = (>>)`).

This option is off by default.

-Wnoncanonical-monadfail-instances

Warn if noncanonical Monad or MonadFail 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 `fail` is defined it must be canonical (i.e. `fail = Control.Monad.Fail.fail`).

Moreover, in MonadFail instance declarations:

- Warn if `fail` is defined backwards (i.e. `fail = Control.Monad.fail`).

See also [-Wmissing-monadfail-instances](#) (page 78).

This option is off by default.

-Wnoncanonical-monoid-instances

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`).

This warning is off by default. However, it is part of the [-Wcompat](#) (page 75) option group.

-Wmissing-monadfail-instances

Warn when a failable pattern is used in a `do`-block that does not have a `MonadFail` instance.

See also *-Wnoncanonical-monadfail-instances* (page 78).

Being part of the *-Wcompat* (page 75) option group, this warning is off by default, but will be switched on in a future GHC release, as part of the [MonadFail Proposal \(MFP\)](#).

-Wsemigroup

Warn when definitions are in conflict with the future inclusion of `Semigroup` into the standard typeclasses.

1. Instances of `Monoid` should also be instances of `Semigroup`
2. The `Semigroup` operator (`<>`) will be in `Prelude`, which clashes with custom local definitions of such an operator

Being part of the *-Wcompat* (page 75) option group, this warning is off by default, but will be switched on in a future GHC release.

-Wdeprecated-flags

Causes a warning to be emitted when a deprecated command-line flag is used.

This option is on by default.

-Wunsupported-calling-conventions

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

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

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

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

Causes a warning to be emitted if a literal will overflow, e.g. `300 :: Word8`.

-Wempty-enumerations

Causes a warning to be emitted if an enumeration is empty, e.g. [5 .. 3].

-Wduplicate-constraints

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 80).

-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.

This option is on by default. As usual you can suppress it on a per-module basis with *-Wno-redundant-constraints* (page 80). 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

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.

This option is on by default.

-Whi-shadowing

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.

-Widentities

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-prelude

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 `-XNoImplicitPrelude` (page 256) or a `LANGUAGE NoImplicitPrelude` pragma).

Note that no warning is given for syntax that implicitly refers to the Prelude, even if `-XNoImplicitPrelude` (page 256) 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).

This warning is off by default.

-Wincomplete-patterns

The option `-Wincomplete-patterns` (page 81) 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 81) 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 75).

-Wincomplete-uni-patterns

The flag `-Wincomplete-uni-patterns` (page 81) is similar to `-Wincomplete-patterns` (page 81), except that it applies only to lambda-expressions and pattern bindings, constructs that only allow a single pattern:

```
h = \[] -> 2
Just k = f y
```

-fmax-pmcheck-iterations={n}

Default 2000000

Sets how many iterations of the pattern-match checker will perform before giving up. This limit is to catch cases where pattern-match checking might be excessively costly (due to the exponential complexity of coverage checking in the general case). It typically shouldn't be necessary to set this unless GHC informs you that it has exceeded the pattern match checker's iteration limit (in which case you may want to consider refactoring your pattern match, for the sake of future readers of your code).

-Wincomplete-record-updates

The function `f` below will fail when applied to `Bar`, so the compiler will emit a warning about this when `-Wincomplete-record-updates` (page 81) 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.

-Wmissing-fields

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 8.4.1

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 82) 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

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 82) 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

This option is on by default, and warns you whenever an instance declaration is missing one or more methods, and the corresponding class declaration has no default declaration for them.

The warning is suppressed if the method name begins with an underscore. Here's an example where this is useful:

```
class C a where
  _simpleFn :: a -> String
  complexFn :: a -> a -> String
  complexFn x y = ... _simpleFn ...
```

The idea is that: (a) users of the class will only call `complexFn`; never `_simpleFn`; and (b) instance declarations can define either `complexFn` or `_simpleFn`.

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 422).

-Wmissing-signatures

If you would like GHC to check that every top-level function/value has a type signature,

use the *-Wmissing-signatures* (page 82) option. As part of the warning GHC also reports the inferred type. The option is off by default.

-Wmissing-exported-sigs

This option is now deprecated in favour of *-Wmissing-exported-signatures* (page 83).

-Wmissing-exported-signatures

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 83) option. This option takes precedence over *-Wmissing-signatures* (page 82). As part of the warning GHC also reports the inferred type. The option is off by default.

-Wmissing-local-sigs

This option is now deprecated in favour of *-Wmissing-local-signatures* (page 83).

-Wmissing-local-signatures

If you use the *-Wmissing-local-signatures* (page 83) 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

If you would like GHC to check that every pattern synonym has a type signature, use the *-Wmissing-pattern-synonym-signatures* (page 83) option. If this option is used in conjunction with *-Wmissing-exported-signatures* (page 83) then only exported pattern synonyms must have a type signature. GHC also reports the inferred type. This option is off by default.

-Wname-shadowing

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

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 162) for details.

The flag *-Worphans* (page 83) warns about user-written orphan rules or instances.

-Woverlapping-patterns

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.

-W`simplifiable-class-constraints`

Since 8.2

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
```

This option is on by default. As usual you can suppress it on a per-module basis with *-Wno-simplifiable-class-constraints* (page 84).

-W`tabs`

Have the compiler warn if there are tabs in your source file.

-W`type-defaults`

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.

This warning is off by default.

-W`monomorphism-restriction`

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.

This warning is off by default.

-W`unsupported-llvm-version`

Warn when using *-fllvm* (page 186) with an unsupported version of LLVM.

-W`unticked-promoted-constructors`

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 warning is enabled by default in *-Wall* (page 75) mode.

-W`unused-binds`

Report any function definitions (and local bindings) which are unused. An alias for

- [-Wunused-top-binds](#) (page 85)
- [-Wunused-local-binds](#) (page 85)
- [-Wunused-pattern-binds](#) (page 85)

-Wunused-top-binds

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

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

Warn if a pattern binding binds no variables at all, unless it is a lone wild-card pattern, or a banded pattern. For example:

```
Just _ = rhs3 -- Warning: unused pattern binding
(_,_) = rhs4 -- Warning: unused pattern binding
_ = rhs3     -- No warning: lone wild-card pattern
!() = rhs4   -- No warning: banded 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 $_v = \text{rhs3}$, which elicits no warning; and they can be useful to add a type constraint, e.g. $_ = x :: \text{Int}$. A banded pattern (see [Bang patterns and Strict Haskell](#) (page 410)) is *not* a no-op, because it forces evaluation, and is useful as an alternative to `seq`.

-Wunused-imports

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

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 85) 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 86) flag.

-Wunused-do-bind

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

Report all unused 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. The warning is suppressed if the type variable name begins with an underscore, like so:

```
type instance F _x _y = []
```

Unlike *-Wunused-matches* (page 85), *-Wunused-type-patterns* (page 86) is not implied by *-Wall* (page 75). 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

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.

-Wwrong-do-bind

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

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 435).

-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

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 237) 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 87) warns about record fields that could fail when accessed via a lacking constructor. The function `f` below will fail when applied to `Bar`, so the compiler will emit a warning at its definition when `-Wpartial-fields` (page 87) is enabled.

The warning is suppressed if the field name begins with an underscore.

```
data Foo = Foo { f :: Int } | Bar
```

If you’re feeling really paranoid, the `-dcore-lint` (page 202) option is a good choice. It turns on heavyweight intra-pass sanity-checking within GHC. (It checks GHC’s sanity, not yours.)

6.3 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.

6.3.1 `-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.

`-Odph`

Enables all `-O2` optimisation, sets `-fmax-simplifier-iterations=20` and `-fsimplifier-phases=3`. Designed for use with *Data Parallel Haskell (DPH)* (page 449).

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 69), then stand back in amazement.

6.3.2 `-f*`: platform-independent flags

These flags turn on and off individual optimisations. Flags marked as on by default are enabled by `-O`, and as such you shouldn’t need to set any of them explicitly. A flag `-fwombat` can be negated by saying `-fno-wombat`.

`-fcase-merge`

Default on

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 on

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 on

Enable call-arity analysis.

-fexitification

Default on

Enables the floating of exit paths out of recursive functions.

-fcmm-elim-common-blocks

Default on

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 on

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.

-fcpr-anal

Default on

Turn on CPR analysis in the demand analyser.

-fcse

Default on

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** on

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

Make dictionaries strict.

-fdmd-tx-dict-sel**Default** on

Use a special demand transformer for dictionary selectors.

-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.

-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 99) 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 506).

-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** on

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** on

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. When optimisation in on, and `-fno-full-laziness` is not given, some transformations that increase sharing are performed, such as extracting repeated computations from a loop. These are the same transformations that a fully lazy implementation would do, the difference is that GHC doesn't consistently apply full-laziness, so don't rely on it.

-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.

-fignore-asserts**Default** on

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 417))).

-fignore-interface-pragmas**Default** 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.

-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 94) 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 [Trac wiki page](#).

-fliberate-case

Default off but enabled with *-O2* (page 88).

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 94)) but for free variables rather than arguments.

-fliberate-case-threshold=<n>

Default 2000

Set the size threshold for the liberate-case transformation.

-floopification

Default on

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

Instructs GHC to use the platform's native vector registers to pass vector arguments during function calls. As with all vector support, this requires *-fllvm* (page 186).

-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 overhead. 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-bindings* gives an unlimited number. Syntactically top-level bindings are also usually excluded (since they may be numerous), but *-fno-max-relevant-bindings* includes them too.

-fmax-valid-substitutions=<n>

Default 6

The type checker sometimes displays a list of valid substitutions for typed holes in error messages, but only up to some maximum number, set by this flag. Turning it off with `-fno-max-valid-substitutions` gives an unlimited number.

-fmax-uncovered-patterns=<n>

Default 4

Maximum number of unmatched patterns to be shown in warnings generated by `-Wincomplete-patterns` (page 81) and `-Wincomplete-uni-patterns` (page 81).

-fmax-simplifier-iterations=<n>

Default 4

Sets the maximal number of iterations for the simplifier.

-fmax-worker-args=<n>

Default 10

If a worker has that many arguments, none will be unpacked anymore.

-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 88), 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 yield points enabled

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 93)). 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 ([Trac #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.

Note that the graph colouring allocator is a bit experimental and may fail when faced with code with high register pressure [Trac #8657](#).

-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 94) 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 432)) that don't terminate, or (less satisfactorily) if you code up recursion through data types (*Bugs in GHC* (page 506)). 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.

-fspec-constr**Default** off but enabled by `-O2` (page 88).

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 a argument is strict in the recursive call to itself but not on the initial entry. As 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 on

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 96) is set imported functions that have an `INLINABLE` pragma (*INLINABLE pragma* (page 424)) 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 90) if you want to ensure all calls are specialised.

-fcross-module-specialise**Default** on

Specialise `INLINABLE` (*INLINABLE pragma* (page 424)) 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.

-fsolve-constant-dicts**Default** on

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](#)

-fstrictness**Default** on

Switch on the strictness analyser. The implementation is described in the paper ‘[Theory and Practice of Demand Analysis in Haskell](https://www.microsoft.com/en-us/research/wp-content/uploads/2017/03/demand-jfp-draft.pdf)’ <<https://www.microsoft.com/en-us/research/wp-content/uploads/2017/03/demand-jfp-draft.pdf>>’.

The strictness analyser figures out when arguments and variables in a function can be treated ‘strictly’ (that is they are always evaluated in the function at some point). This allow GHC to apply certain optimisations such as unboxing that otherwise don’t apply as they change the semantics of the program when applied to lazy arguments.

-fstrictness-before={n}

Run an additional strictness analysis before simplifier phase {n}.

-funbox-small-strict-fields

Default on

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 429)) 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 236)) 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 430)).

Note that for consistency Double, Word64, and Int64 constructor fields are unpacked on 32-bit platforms, even though they are technically 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 429)).

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 430)).

Alternatively you can use -funbox-small-strict-fields (page 97) 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:

- 1.nothing larger than this will be inlined (unless it has an `INLINE` pragma)
- 2.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 98) 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-keenness-factor={n}

Default 1.5

How eager should the compiler be to inline functions?

-funfolding-use-threshold={n}

Default 60

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 97) 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.

-fvectorisation-avoidance

Default on

Part of *Data Parallel Haskell (DPH)* (page 449).

Enable the *vectorisation* avoidance optimisation. This optimisation only works when used in combination with the `-fvectorise` transformation.

While vectorisation of code using DPH is often a big win, it can also produce worse results for some kinds of code. This optimisation modifies the vectorisation transformation to try to determine if a function would be better off unvectorised and if so, do just that.

-fvectorise

Default off

Part of *Data Parallel Haskell (DPH)* (page 449).

Enable the *vectorisation* optimisation transformation. This optimisation transforms the nested data parallelism code of programs using DPH into flat data parallelism. Flat data parallel programs should have better load balancing, enable SIMD parallelism and friendlier cache behaviour.

6.4 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 189) option (see *Options affecting linking* (page 187)). This provides two benefits:

- It enables the *-N {x}* (page 100) to be used, which allows threads to run in parallelism on a multi-processor or multi-core machine. See *Using SMP parallelism* (page 99).
- 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 468).

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).

6.5 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 *Concurrent and Parallel Haskell* (page 447) we describe the language features that affect parallelism.

6.5.1 Compile-time options for SMP parallelism

In order to make use of multiple CPUs, your program must be linked with the *-threaded* (page 189) option (see *Options affecting linking* (page 187)). 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 90) 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 90) flag.

6.5.2 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 100) options.

-N <x>

-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 468)) another OS thread can take over that capability.

Normally `<x>` should be chosen to match the number of CPU cores on the machine¹. 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.

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 134)).

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.

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.

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`.

6.5.3 Hints for using SMP parallelism

Add the `-s [[file]]` (page 140) 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 100) 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 134)), 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.

6.6 Flag reference

This section is a quick-reference for GHC’s command-line flags. For each flag, we also list its static/dynamic status (see [Static, Dynamic, and Mode options](#) (page 65)), and the flag’s opposite (if available).

6.6.1 Verbosity options

More details in [Verbosity options](#) (page 69)

Flag	Description	Type	Reverse
<code>-fdiagnostics-color={always auto never}</code> (page 72)	Use colors in error messages	dynamic	
<code>-fdiagnostics-show-caret</code> (page 73)	Whether to show snippets of original source code	dynamic	<code>-fno-diagnostics-show-caret</code>
<code>-ferror-spans</code> (page 73)	Output full span in error messages	dynamic	
Continued on next page			

Table 6.1 – continued from previous page

Flag	Description	Type	Reverse
<i>-fhide-source-paths</i> (page 70)	hide module source and object paths	dynamic	
<i>-fprint-equality-relations</i> (page 71)	Distinguish between equality relations when printing	dynamic	<i>-fno-print-equality-relations</i>
<i>-fprint-expanded-synonyms</i> (page 71)	In type errors, also print type-synonym-expanded types.	dynamic	<i>-fno-print-expanded-synonyms</i>
<i>-fprint-explicit-coercions</i> (page 71)	Print coercions in types	dynamic	<i>-fno-print-explicit-coercions</i>
<i>-fprint-explicit-foralls</i> (page 70)	Print explicit forall quantification in types. See also <i>-XExplicitForAll</i> (page 367)	dynamic	<i>-fno-print-explicit-foralls</i>
<i>-fprint-explicit-kinds</i> (page 71)	Print explicit kind foralls and kind arguments in types. See also <i>-XKindSignatures</i> (page 369)	dynamic	<i>-fno-print-explicit-kinds</i>
<i>-fprint-explicit-runtime-rep</i> (page 362)	Print RuntimeRep variables in types which are runtime-representation polymorphic.	dynamic	<i>-fno-print-explicit-runtime-reps</i>
<i>-fprint-explicit-runtime-reps</i> (page 71)	Print RuntimeRep variables in types which are runtime-representation polymorphic.	dynamic	<i>-fno-print-explicit-runtime-reps</i>
<i>-fprint-potential-instances</i> (page 70)	display all available instances in type error messages	dynamic	<i>-fno-print-potential-instances</i>
<i>-fprint-typechecker-elaboration</i> (page 72)	Print extra information from typechecker.	dynamic	<i>-fno-print-typechecker-elaboration</i>
<i>-fprint-unicode-syntax</i> (page 70)	Use unicode syntax when printing expressions, types and kinds. See also <i>-XUnicodeSyntax</i> (page 240)	dynamic	<i>-fno-print-unicode-syntax</i>
<i>-fshow-hole-constraints</i> (page 387)	Show constraints when reporting typed holes	dynamic	
<i>-Rghc-timing</i> (page 73)	Summarise timing stats for GHC (same as +RTS -tstderr).	dynamic	
<i>-v</i> (page 69)	verbose mode (equivalent to <i>-v3</i>)	dynamic	
<i>-v<n></i> (page 69)	set verbosity level	dynamic	

6.6.2 Alternative modes of operation

More details in *Modes of operation* (page 66)

Flag	Description	Type	Reverse
<code>--frontend {module}</code> (page 66)	run GHC with the given frontend plugin; see <i>Frontend plugins</i> (page 480) for details.	mode	
<code>--help</code> (page 66), <code>-?</code> (page 66)	Display help	mode	
<code>--info</code> (page 67)	display information about the compiler	mode	
<code>--interactive</code> (page 66)	Interactive mode - normally used by just running <code>ghci</code> ; see <i>Using GHCi</i> (page 17) for details.	mode	
<code>--make</code> (page 66)	Build a multi-module Haskell program, automatically figuring out dependencies. Likely to be much easier, and faster, than using <code>make</code> ; see <i>Using ghc -make</i> (page 67) for details.	mode	
<code>--mk-dll</code> (page 66)	DLL-creation mode (Windows only)	mode	
<code>--numeric-version</code> (page 67)	display GHC version (numeric only)	mode	
<code>--print-libdir</code> (page 67)	display GHC library directory	mode	
<code>--show-iface {file}</code> (page 66)	display the contents of an interface file.	mode	
<code>--show-options</code> (page 67)	display the supported command line options	mode	
<code>--supported-extensions</code> (page 67), <code>--supported-languages</code> (page 67)	display the supported language extensions	mode	
<code>--version</code> (page 67), <code>-V</code> (page 67)	display GHC version	mode	
<code>-e {expr}</code> (page 66)	Evaluate <code>expr</code> ; see <i>Expression evaluation mode</i> (page 68) for details.	mode	
<code>-M</code> (page 66)	generate dependency information suitable for use in a Makefile; see <i>Dependency generation</i> (page 161) for details.	mode	

6.6.3 Which phases to run

More details in *Batch compiler mode* (page 68)

Flag	Description	Type	Reverse
<code>-C</code> (page 66)	Stop after generating C (.hc file)	mode	
<code>-c</code> (page 66)	Stop after generating object (.o) file	mode	
<code>-E</code> (page 66)	Stop after preprocessing (.hspp file)	mode	
<code>-F</code> (page 185)	Enable the use of a <i>pre-processor</i> (page 185) (set with <code>-pgmF {cmd}</code> (page 182))	dynamic	
<code>-S</code> (page 66)	Stop after generating assembly (.s file)	mode	
<code>-x {suffix}</code> (page 69)	Override default behaviour for source files	dynamic	

6.6.4 Redirecting output

More details in *Redirecting the compilation output(s)* (page 149)

Flag	Description	Type	Reverse
<code>--exclude-module={file}</code> (page 162)	Regard {file} as “stable”; i.e., exclude it from having dependencies on it.	dynamic	
<code>-ddump-mod-cycles</code> (page 162)	Dump module cycles	dynamic	
<code>-dep-makefile {file}</code> (page 162)	Use {file} as the makefile	dynamic	
<code>-dep-suffix {suffix}</code> (page 162)	Make dependencies that declare that files with suffix <code>.{suf}{osuf}</code> depend on interface files with suffix <code>.{suf}hi</code>	dynamic	
<code>-dumpdir {dir}</code> (page 150)	redirect dump files	dynamic	
<code>-hcsuf {suffix}</code> (page 150)	set the suffix to use for intermediate C files	dynamic	
<code>-hidir {dir}</code> (page 150)	set directory for interface files	dynamic	
<code>-hisuf {suffix}</code> (page 150)	set the suffix to use for interface files	dynamic	
<code>-include-pkg-deps</code> (page 162)	Regard modules imported from packages as unstable	dynamic	
<code>-o {file}</code> (page 149)	set output filename	dynamic	
<code>-odir {dir}</code> (page 149)	set directory for object files	dynamic	

Continued on next page

Table 6.4 – continued from previous page

Flag	Description	Type	Reverse
<code>-ohi {file}</code> (page 149)	set the filename in which to put the interface	dynamic	
<code>-osuf {suffix}</code> (page 150)	set the output file suffix	dynamic	
<code>-outputdir {dir}</code> (page 150)	set output directory	dynamic	
<code>-stubdir {dir}</code> (page 150)	redirect FFI stub files	dynamic	

6.6.5 Keeping intermediate files

More details in *Keeping Intermediate Files* (page 150)

Flag	Description	Type	Reverse
<code>-keep-hc-file</code> (page 150), <code>-keep-hc-files</code> (page 150)	Retain intermediate .hc files.	dynamic	
<code>-keep-hi-files</code> (page 150)	Retain intermediate .hi files (the default).	dynamic	<code>-no-keep-hi-files</code>
<code>-keep-llvm-file</code> (page 150), <code>-keep-llvm-files</code> (page 150)	Retain intermediate LLVM .ll files. Implies <code>-fllvm</code> (page 186).	dynamic	
<code>-keep-o-files</code> (page 151)	Retain intermediate .o files (the default).	dynamic	<code>-no-keep-o-files</code>
<code>-keep-s-file</code> (page 151), <code>-keep-s-files</code> (page 151)	Retain intermediate .s files.	dynamic	
<code>-keep-tmp-files</code> (page 151)	Retain all intermediate temporary files.	dynamic	

6.6.6 Temporary files

More details in *Redirecting temporary files* (page 151)

Flag	Description	Type	Reverse
<code>-tmpdir {dir}</code> (page 151)	set the directory for temporary files	dynamic	

6.6.7 Finding imports

More details in *The search path* (page 148)

Flag	Description	Type	Reverse
<code>-i</code> (page 148)	Empty the import directory list	dynamic/ :set	
Continued on next page			

Table 6.7 – continued from previous page

Flag	Description	Type	Reverse
<code>-i{dir}[:{dir}]*</code> (page 148)	add {dir}, {dir2}, etc. to import path	dynamic/ :set	

6.6.8 Interface file options

More details in *Other options related to interface files* (page 151)

Flag	Description	Type	Reverse
<code>--show-iface {file}</code> (page 66)	See <i>Modes of operation</i> (page 66).	mode	
<code>-ddump-hi</code> (page 151)	Dump the new interface to stdout	dynamic	
<code>-ddump-hi-diffs</code> (page 151)	Show the differences vs. the old interface	dynamic	
<code>-ddump-minimal-imports</code> (page 151)	Dump a minimal set of imports	dynamic	

6.6.9 Recompilation checking

More details in *The recompilation checker* (page 152)

Flag	Description	Type	Reverse
<code>-fforce-recomp</code> (page 152)	Turn off recompilation checking. This is implied by any <code>-ddump-X</code> option when compiling a single file (i.e. when using <code>-c</code> (page 66)).	dynamic	<code>-fno-force-recomp</code>
<code>-fignore-hpc-changes</code> (page 152)	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	<code>-fno-ignore-hpc-changes</code>
<code>-fignore-optim-changes</code> (page 152)	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	<code>-fno-ignore-optim-changes</code>

6.6.10 Interactive-mode options

More details in *The .ghci and .haskeline files* (page 56)

Flag	Description	Type	Reverse
<code>-fbreak-on-error</code> (page 41)	<i>Break on uncaught exceptions and errors</i> (page 40)	dynamic	<code>-fno-break-on-error</code>
<code>-fbreak-on-exception</code> (page 40)	<i>Break on any exception thrown</i> (page 40)	dynamic	<code>-fno-break-on-exception</code>
<code>-fghci-hist-size={n}</code> (page 40)	Set the number of entries GHCi keeps for <code>:history</code> . See <i>The GHCi Debugger</i> (page 33).	dynamic	
<code>-flocal-ghci-history</code> (page 42)	Use current directory for the GHCi command history file <code>.ghci-history</code> .	dynamic	<code>-fno-local-ghci-history</code>
<code>-fprint-bind-result</code> (page 22)	<i>Turn on printing of binding results in GHCi</i> (page 22)	dynamic	<code>-fno-print-bind-result</code>
<code>-fshow-loaded-modules</code> (page 18)	Show the names of modules that GHCi loaded after a <code>:load</code> (page 48) command.	dynamic	
<code>-ghci-script</code> (page 57)	Read additional <code>.ghci</code> files	dynamic	
<code>-ignore-dot-ghci</code> (page 57)	Disable reading of <code>.ghci</code> files	dynamic	
<code>-interactive-print {expr}</code> (page 32)	<i>Select the function to use for printing evaluated expressions in GHCi</i> (page 31)	dynamic	

6.6.11 Packages

More details in *Packages* (page 163)

Flag	Description	Type	Reverse
<code>-clear-package-db</code> (page 169)	Clear the package db stack.	dynamic	
<code>-distrust {pkg}</code> (page 457)	Expose package <code>{pkg}</code> and set it to be distrusted. See <i>Safe Haskell</i> (page 449).	dynamic/ :set	
<code>-distrust-all-packages</code> (page 457)	Distrust all packages by default. See <i>Safe Haskell</i> (page 449).	dynamic/ :set	
<code>-fpackage-trust</code> (page 459)	Enable <i>Safe Haskell</i> (page 449) trusted package requirement for trustworthy modules.	dynamic	
<code>-global-package-db</code> (page 169)	Add the global package db to the stack.	dynamic	
<code>-hide-all-packages</code> (page 166)	Hide all packages by default	dynamic	

Continued on next page

Table 6.11 – continued from previous page

Flag	Description	Type	Reverse
<code>-hide-package <pkg></code> (page 166)	Hide package <pkg>	dynamic/ :set	
<code>-ignore-package <pkg></code> (page 166)	Ignore package <pkg>	dynamic/ :set	
<code>-no-auto-link-packages</code> (page 166)	Don't automatically link in the base and rts packages.	dynamic	
<code>-no-global-package-db</code> (page 169)	Remove the global package db from the stack.	dynamic	
<code>-no-user-package-db</code> (page 169)	Remove the user's package db from the stack.	dynamic	
<code>-package <pkg></code> (page 165)	Expose package <pkg>	dynamic/ :set	
<code>-package-db <file></code> (page 169)	Add <file> to the package db stack.	dynamic	
<code>-package-env <file> <name></code> (page 170)	Use the specified package environment.	dynamic	
<code>-package-id <unit-id></code> (page 166)	Expose package by id <unit-id>	dynamic/ :set	
<code>-this-unit-id <unit-id></code> (page 166)	Compile to be part of unit (i.e. package) <unit-id>	dynamic	
<code>-trust <pkg></code> (page 457)	Expose package <pkg> and set it to be trusted. See <i>Safe Haskell</i> (page 449).	dynamic/ :set	
<code>-user-package-db</code> (page 169)	Add the user's package db to the stack.	dynamic	

6.6.12 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 *Language options* (page 233).

6.6.13 Warnings

More details in *Warnings and sanity-checking* (page 74)

Flag	Description	Type	Reverse
<code>-fdefer-out-of-scope-variables</code> (page 77)	Convert variable out of scope variables errors into warnings. Implied by <code>-fdefer-type-errors</code> (page 76). See also <code>-Wdeferred-out-of-scope-variables</code> (page 77).	dynamic	<code>-fno-defer-out-of-scope-variables</code>
Continued on next page			

Table 6.12 – continued from previous page

Flag	Description	Type	Reverse
<code>-fdefer-type-errors</code> (page 76)	Turn type errors into warnings, <i>deferring the error until runtime</i> (page 393). Implies <code>-fdefer-typed-holes</code> (page 76) and <code>-fdefer-out-of-scope-variables</code> (page 77). See also <code>-Wdeferred-type-errors</code> (page 76)	dynamic	<code>-fno-defer-type-errors</code>
<code>-fdefer-typed-holes</code> (page 76)	Convert <i>typed hole</i> (page 385) errors into warnings, <i>deferring the error until runtime</i> (page 393). Implied by <code>-fdefer-type-errors</code> (page 76). See also <code>-Wtyped-holes</code> (page 76).	dynamic	<code>-fno-defer-typed-holes</code>
<code>-fhelpful-errors</code> (page 77)	Make suggestions for misspelled names.	dynamic	<code>-fno-helpful-errors</code>
<code>-fmax-pmcheck-iterations={n}</code> (page 81)	the iteration limit for the pattern match checker	dynamic	
<code>-fshow-warning-groups</code> (page 76)	show which group an emitted warning belongs to.	dynamic	<code>-fno-show-warning-groups</code>
<code>-W</code> (page 75)	enable normal warnings	dynamic	<code>-w</code> (page 75)
<code>-w</code> (page 75)	disable all warnings	dynamic	
<code>-Wall</code> (page 75)	enable almost all warnings (details in <i>Warnings and sanity-checking</i> (page 74))	dynamic	<code>-w</code> (page 75)
<code>-Wall-missed-specialisations</code> (page 77)	warn when specialisation of any overloaded function fails.	dynamic	<code>-Wno-all-missed-specialisations</code>
<code>-Wamp</code> (page 78)	<i>(deprecated)</i> warn on definitions conflicting with the Applicative-Monad Proposal (AMP)	dynamic	<code>-Wno-amp</code>
<code>-Wcompat</code> (page 75)	enable future compatibility warnings (details in <i>Warnings and sanity-checking</i> (page 74))	dynamic	<code>-Wno-compat</code> (page 75)
<code>-Wcpp-undef</code> (page 87)	warn on uses of the <code>#if</code> directive on undefined identifiers	dynamic	
Continued on next page			

Table 6.12 – continued from previous page

Flag	Description	Type	Reverse
<i>-Wdeferred-out-of-scope-variables</i> (page 77)	Report warnings when variable out-of-scope errors are <i>deferred until runtime</i> (page 393). See <i>-fdefer-out-of-scope-variables</i> (page 77).	dynamic	-Wno-deferred-out-of-scope-variables
<i>-Wdeferred-type-errors</i> (page 76)	Report warnings when <i>deferred type errors</i> (page 393) are enabled. This option is enabled by default. See <i>-fdefer-type-errors</i> (page 76).	dynamic	-Wno-deferred-type-errors
<i>-Wdeprecated-flags</i> (page 79)	warn about uses of commandline flags that are deprecated	dynamic	-Wno-deprecated-flags
<i>-Wdeprecations</i> (page 78)	warn about uses of functions & types that have warnings or deprecated pragmas. Alias for <i>-Wwarnings-deprecations</i> (page 77)	dynamic	-Wno-deprecations
<i>-Wdodgy-exports</i> (page 79)	warn about dodgy exports	dynamic	-Wno-dodgy-exports
<i>-Wdodgy-foreign-imports</i> (page 79)	warn about dodgy foreign imports	dynamic	-Wno-dodgy-foreign-import
<i>-Wdodgy-imports</i> (page 79)	warn about dodgy imports	dynamic	-Wno-dodgy-imports
<i>-Wduplicate-constraints</i> (page 80)	warn when a constraint appears duplicated in a type signature	dynamic	-Wno-duplicate-constraints
<i>-Wduplicate-exports</i> (page 80)	warn when an entity is exported multiple times	dynamic	-Wno-duplicate-exports
<i>-Wempty-enumerations</i> (page 79)	warn about enumerations that are empty	dynamic	-Wno-empty-enumerations
<i>-Werror</i> (page 75)	make warnings fatal	dynamic	<i>-Wwarn</i> (page 76)
<i>-Weverything</i> (page 75)	enable all warnings supported by GHC	dynamic	
<i>-Whi-shadowing</i> (page 80)	warn when a .hi file in the current directory shadows a library	dynamic	-Wno-hi-shadowing
<i>-Widentities</i> (page 80)	warn about uses of Prelude numeric conversions that are probably the identity (and hence could be omitted)	dynamic	-Wno-identities
Continued on next page			

Table 6.12 – continued from previous page

Flag	Description	Type	Reverse
<i>-Wimplicit-prelude</i> (page 81)	warn when the Prelude is implicitly imported	dynamic	-Wno-implicit-prelude
<i>-Wincomplete-patterns</i> (page 81)	warn when a pattern match could fail	dynamic	-Wno-incomplete-patterns
<i>-Wincomplete-record-updates</i> (page 81)	warn when a record update could fail	dynamic	-Wno-incomplete-record-updates
<i>-Wincomplete-uni-patterns</i> (page 81)	warn when a pattern match in a lambda expression or pattern binding could fail	dynamic	-Wno-incomplete-uni-patterns
<i>-Winline-rule-shadowing</i> (page 87)	Warn if a rewrite RULE might fail to fire because the function might be inlined before the rule has a chance to fire. See <i>How rules interact with INLINE/NOINLINE pragmas</i> (page 435).	dynamic	-Wno-inline-rule-shadowing
<i>-Wmissed-specialisations</i> (page 77)	warn when specialisation of an imported, overloaded function fails.	dynamic	-Wno-missed-specialisations
<i>-Wmissing-export-lists</i> (page 82)	warn when a module declaration does not explicitly list all exports	dynamic	-fnowarn-missing-export-lists
<i>-Wmissing-exported-signatures</i> (page 83)	warn about top-level functions without signatures, only if they are exported. takes precedence over -Wmissing-signatures	dynamic	-Wno-missing-exported-signatures
<i>-Wmissing-exported-sigs</i> (page 83)	<i>(deprecated)</i> warn about top-level functions without signatures, only if they are exported. takes precedence over -Wmissing-signatures	dynamic	-Wno-missing-exported-sigs
<i>-Wmissing-fields</i> (page 82)	warn when fields of a record are uninitialised	dynamic	-Wno-missing-fields
<i>-Wmissing-home-modules</i> (page 87)	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
Continued on next page			

Table 6.12 – continued from previous page

Flag	Description	Type	Reverse
<i>-Wmissing-import-lists</i> (page 82)	warn when an import declaration does not explicitly list all the names brought into scope	dynamic	<i>-fnowarn-missing-import-lists</i>
<i>-Wmissing-local-signatures</i> (page 83)	warn about polymorphic local bindings without signatures	dynamic	<i>-Wno-missing-local-signatures</i>
<i>-Wmissing-local-sigs</i> (page 83)	<i>(deprecated)</i> warn about polymorphic local bindings without signatures	dynamic	<i>-Wno-missing-local-sigs</i>
<i>-Wmissing-methods</i> (page 82)	warn when class methods are undefined	dynamic	<i>-Wno-missing-methods</i>
<i>-Wmissing-monadfail-instances</i> (page 78)	Warn when a failable pattern is used in a do-block that does not have a MonadFail instance.	dynamic	<i>-Wno-missing-monadfail-instances</i>
<i>-Wmissing-pattern-synonym-signatures</i> (page 83)	warn when pattern synonyms do not have type signatures	dynamic	<i>-Wno-missing-pattern-synonym-signatures</i>
<i>-Wmissing-signatures</i> (page 82)	warn about top-level functions without signatures	dynamic	<i>-Wno-missing-signatures</i>
<i>-Wmonomorphism-restriction</i> (page 84)	warn when the Monomorphism Restriction is applied	dynamic	<i>-Wno-monomorphism-restriction</i>
<i>-Wname-shadowing</i> (page 83)	warn when names are shadowed	dynamic	<i>-Wno-name-shadowing</i>
<i>-Wno-compat</i> (page 75)	Disables all warnings enabled by <i>-Wcompat</i> (page 75).	dynamic	<i>-Wcompat</i> (page 75)
<i>-Wnoncanonical-monad-instances</i> (page 78)	warn when Applicative or Monad instances have noncanonical definitions of return, pure, (>>), or (*>). See flag description in <i>Warnings and sanity-checking</i> (page 74) for more details.	dynamic	<i>-Wno-noncanonical-monad-instances</i>
<i>-Wnoncanonical-monadfail-instances</i> (page 78)	warn when Monad or MonadFail instances have noncanonical definitions of fail. See flag description in <i>Warnings and sanity-checking</i> (page 74) for more details.	dynamic	<i>-Wno-noncanonical-monadfail-instances</i>
Continued on next page			

Table 6.12 – continued from previous page

Flag	Description	Type	Reverse
<i>-Wnoncanonical-monoid-instances</i> (page 78)	warn when Semigroup or Monoid instances have noncanonical definitions of (<code><></code>) or <code>mappend</code> . See flag description in <i>Warnings and sanity-checking</i> (page 74) for more details.	dynamic	<i>-Wno-noncanonical-monoid-instances</i>
<i>-Worphans</i> (page 83)	warn when the module contains <i>orphan instance declarations or rewrite rules</i> (page 162)	dynamic	<i>-Wno-orphans</i>
<i>-Woverflowed-literals</i> (page 79)	warn about literals that will overflow their type	dynamic	<i>-Wno-overflowed-literals</i>
<i>-Woverlapping-patterns</i> (page 83)	warn about overlapping patterns	dynamic	<i>-Wno-overlapping-patterns</i>
<i>-Wpartial-fields</i> (page 87)	warn when defining a partial record field.	dynamic	<i>-Wno-partial-fields</i>
<i>-Wpartial-type-signatures</i> (page 77)	warn about holes in partial type signatures when <i>-XPartialTypeSignatures</i> (page 387) is enabled. Not applicable when <i>-XPartialTypesignatures</i> is not enabled, in which case errors are generated for such holes. See <i>Partial Type Signatures</i> (page 387).	dynamic	<i>-Wno-partial-type-signatures</i>
<i>-Wredundant-constraints</i> (page 80)	Have the compiler warn about redundant constraints in type signatures.	dynamic	<i>-Wno-redundant-constraints</i>
<i>-Wsafe</i> (page 459)	warn if the module being compiled is regarded to be safe.	dynamic	<i>-Wno-safe</i>
<i>-Wsemigroup</i> (page 79)	warn when a Monoid is not Semigroup, and on non-Semigroup definitions of (<code><></code>)?	dynamic	<i>-Wno-semigroup</i>
<i>-Wsimplifiable-class-constraints</i> (page 84)	Warn about class constraints in a type signature that can be simplified using a top-level instance declaration.	dynamic	<i>-Wno-overlapping-patterns</i>
<i>-Wtabs</i> (page 84)	warn if there are tabs in the source file	dynamic	<i>-Wno-tabs</i>
Continued on next page			

Table 6.12 – continued from previous page

Flag	Description	Type	Reverse
<i>-Wtrustworthy-safe</i> (page 459)	warn if the module being compiled is marked as <i>Trustworthy</i> (page 458) but it could instead be marked as <i>Safe</i> (page 458), a more informative bound.	dynamic	-Wno-safe
<i>-Wtype-defaults</i> (page 84)	warn when defaulting happens	dynamic	-Wno-type-defaults
<i>-Wtyped-holes</i> (page 76)	Report warnings when <i>typed hole</i> (page 385) errors are <i>deferred until runtime</i> (page 393). See <i>-fdefer-typed-holes</i> (page 76).	dynamic	-Wno-typed-holes
<i>-Wunbanged-strict-patterns</i> (page 87)	warn on pattern bind of unlifted variable that is neither bare nor banged	dynamic	-Wno-unbanged-strict-patterns
<i>-Wunrecognised-pragmas</i> (page 77)	warn about uses of pragmas that GHC doesn't recognise	dynamic	-Wno-unrecognised-pragmas
<i>-Wunrecognised-warning-flags</i> (page 76)	throw a warning when an unreconised -W... flag is encountered on the command line.	dynamic	-Wno-unrecognised-warning-flags
<i>-Wunsafe</i> (page 459)	warn if the module being compiled is regarded to be unsafe. See <i>Safe Haskell</i> (page 449)	dynamic	-Wno-unsafe
<i>-Wunsupported-calling-conventions</i> (page 79)	warn about use of an unsupported calling convention	dynamic	-Wno-unsupported-calling-conventions
<i>-Wunsupported-llvm-version</i> (page 84)	Warn when using <i>-fllvm</i> (page 186) with an unsupported version of LLVM.	dynamic	-Wno-monomorphism-restriction
<i>-Wunticked-promoted-constructors</i> (page 84)	warn if promoted constructors are not ticked	dynamic	-Wno-unticked-promoted-constructors
<i>-Wunused-binds</i> (page 84)	warn about bindings that are unused. Alias for <i>-Wunused-top-binds</i> (page 85), <i>-Wunused-local-binds</i> (page 85) and <i>-Wunused-pattern-binds</i> (page 85)	dynamic	-Wno-unused-binds
<i>-Wunused-do-bind</i> (page 86)	warn about do bindings that appear to throw away values of types other than ()	dynamic	-Wno-unused-do-bind
Continued on next page			

Table 6.12 – continued from previous page

Flag	Description	Type	Reverse
<i>-Wunused-foralls</i> (page 86)	warn about type variables in user-written forall's that are unused	dynamic	-Wno-unused-foralls
<i>-Wunused-imports</i> (page 85)	warn about unnecessary imports	dynamic	-Wno-unused-imports
<i>-Wunused-local-binds</i> (page 85)	warn about local bindings that are unused	dynamic	-Wno-unused-local-binds
<i>-Wunused-matches</i> (page 85)	warn about variables in patterns that aren't used	dynamic	-Wno-unused-matches
<i>-Wunused-pattern-binds</i> (page 85)	warn about pattern match bindings that are unused	dynamic	-Wno-unused-pattern-binds
<i>-Wunused-top-binds</i> (page 85)	warn about top-level bindings that are unused	dynamic	-Wno-unused-top-binds
<i>-Wunused-type-patterns</i> (page 86)	warn about unused type variables which arise from patterns in type family and data family instances	dynamic	-Wno-unused-type-patterns
<i>-Wwarn</i> (page 76)	make warnings non-fatal	dynamic	<i>-Werror</i> (page 75)
<i>-Wwarnings-deprecations</i> (page 77)	warn about uses of functions & types that have warnings or deprecated pragmas	dynamic	-Wno-warnings-deprecations
<i>-Wwrong-do-bind</i> (page 86)	warn about do bindings that appear to throw away monadic values that you should have bound instead	dynamic	-Wno-wrong-do-bind

6.6.14 Optimisation levels

These options are described in more detail in *Optimisation (code improvement)* (page 87).

See *Individual optimisations* (page 116) for a list of optimisations enabled on level 1 and level 2.

Flag	Description	Type	Reverse
<i>-O</i> (page 88), <i>-O1</i> (page 88)	Enable level 1 optimisations	dynamic	<i>-O0</i> (page 88)
<i>-O0</i> (page 88)	Disable optimisations (default)	dynamic	
<i>-O2</i> (page 88)	Enable level 2 optimisations	dynamic	<i>-O0</i> (page 88)
<i>-Odph</i> (page 88)	Enable level 2 optimisations, set <code>-fmax-simplifier-iterations=20</code> and <code>-fsimplifier-phases=3</code> .	dynamic	

6.6.15 Individual optimisations

These options are described in more detail in *-f**: [platform-independent flags](#) (page 88). 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
-fcall-arity (page 89)	Enable call-arity optimisation. Implied by <code>-O</code> (page 88).	dynamic	<code>-fno-call-arity</code>
-fcase-folding (page 89)	Enable constant folding in case expressions. Implied by <code>-O</code> (page 88).	dynamic	<code>-fno-case-folding</code>
-fcase-merge (page 88)	Enable case-merging. Implied by <code>-O</code> (page 88).	dynamic	<code>-fno-case-merge</code>
-fcmm-elim-common-blocks (page 89)	Enable Cmm common block elimination. Implied by <code>-O</code> (page 88).	dynamic	<code>-fno-cmm-elim-common-blocks</code>
-fcmm-sink (page 89)	Enable Cmm sinking. Implied by <code>-O</code> (page 88).	dynamic	<code>-fno-cmm-sink</code>
-fcpr-anal (page 89)	Turn on CPR analysis in the demand analyser. Implied by <code>-O</code> (page 88).	dynamic	<code>-fno-cpr-anal</code>
-fcross-module-specialise (page 96)	Turn on specialisation of overloaded functions imported from other modules.	dynamic	<code>-fno-cross-module-specialise</code>
-fcse (page 89)	Enable common sub-expression elimination. Implied by <code>-O</code> (page 88).	dynamic	<code>-fno-cse</code>
-fdicts-cheap (page 90)	Make dictionary-valued expressions seem cheap to the optimiser.	dynamic	<code>-fno-dicts-cheap</code>
-fdicts-strict (page 90)	Make dictionaries strict	dynamic	<code>-fno-dicts-strict</code>
-fdmd-tx-dict-sel (page 90)	Use a special demand transformer for dictionary selectors. Always enabled by default.	dynamic	<code>-fno-dmd-tx-dict-sel</code>
-fdo-eta-reduction (page 90)	Enable eta-reduction. Implied by <code>-O</code> (page 88).	dynamic	<code>-fno-do-eta-reduction</code>
-fdo-lambda-eta-expansion (page 90)	Enable lambda eta-expansion. Always enabled by default.	dynamic	<code>-fno-do-lambda-eta-expansion</code>
-feager-blackholing (page 90)	Turn on eager blackholing (page 99)	dynamic	
Continued on next page			

Table 6.14 – continued from previous page

Flag	Description	Type	Reverse
<i>-fenable-rewrite-rules</i> (page 432)	Switch on all rewrite rules (including rules generated by automatic specialisation of overloaded functions). Implied by <i>-O</i> (page 88).	dynamic	<i>-fno-enable-rewrite-rules</i>
<i>-fexcess-precision</i> (page 90)	Enable excess intermediate precision	dynamic	<i>-fno-excess-precision</i>
<i>-fexitification</i> (page 89)	Enables exitification optimisation. Implied by <i>-O</i> (page 88).	dynamic	<i>-fno-exitification</i>
<i>-fexpose-all-unfoldings</i> (page 90)	Expose all unfoldings, even for very large or recursive functions.	dynamic	<i>-fno-expose-all-unfoldings</i>
<i>-ffloat-in</i> (page 91)	Turn on the float-in transformation. Implied by <i>-O</i> (page 88).	dynamic	<i>-fno-float-in</i>
<i>-ffull-laziness</i> (page 91)	Turn on full laziness (floating bindings outwards). Implied by <i>-O</i> (page 88).	dynamic	<i>-fno-full-laziness</i>
<i>-ffun-to-thunk</i> (page 91)	Allow worker-wrapper to convert a function closure into a thunk if the function does not use any of its arguments. Off by default.	dynamic	<i>-fno-fun-to-thunk</i>
<i>-fignore-asserts</i> (page 91)	Ignore assertions in the source. Implied by <i>-O</i> (page 88).	dynamic	<i>-fno-ignore-asserts</i>
<i>-fignore-interface-pragmas</i> (page 91)	Ignore pragmas in interface files. Implied by <i>-O0</i> (page 88) only.	dynamic	<i>-fno-ignore-interface-pragmas</i>
<i>-flate-dmd-anal</i> (page 91)	Run demand analysis again, at the end of the simplification pipeline	dynamic	<i>-fno-late-dmd-anal</i>
<i>-fliberate-case</i> (page 92)	Turn on the liberate-case transformation. Implied by <i>-O2</i> (page 88).	dynamic	<i>-fno-liberate-case</i>
<i>-fliberate-case-threshold={n}</i> (page 92)	<i>default: 2000</i> . Set the size threshold for the liberate-case transformation to {n}	dynamic	<i>-fno-liberate-case-threshold</i>
<i>-fllvm-pass-vectors-in-regs</i> (page 92)	Pass vector value in vector registers for function calls	dynamic	<i>-fno-llvm-pass-vectors-in-regs</i>
<i>-floopification</i> (page 92)	Turn saturated self-recursive tail-calls into local jumps in the generated assembly. Implied by <i>-O</i> (page 88).	dynamic	<i>-fno-loopification</i>
Continued on next page			

Table 6.14 – continued from previous page

Flag	Description	Type	Reverse
<code>-fmax-inline-alloc-size={n}</code> (page 92)	<i>default:</i> 128. Set the maximum size of inline array allocations to {n} bytes (default: 128).	dynamic	
<code>-fmax-inline-memcpy-insns={n}</code> (page 92)	<i>default:</i> 32. Inline memcpy calls if they would generate no more than {n} pseudo instructions.	dynamic	
<code>-fmax-inline-memset-insns={n}</code> (page 92)	<i>default:</i> 32. Inline memset calls if they would generate no more than {n} pseudo instructions	dynamic	
<code>-fmax-relevant-binds={n}</code> (page 92)	<i>default:</i> 6. Set the maximum number of bindings to display in type error messages.	dynamic	<code>-fno-max-relevant-bindings</code>
<code>-fmax-simplifier-iterations={n}</code> (page 93)	<i>default:</i> 4. Set the max iterations for the simplifier.	dynamic	
<code>-fmax-uncovered-patterns={n}</code> (page 93)	<i>default:</i> 4. Set the maximum number of patterns to display in warnings about non-exhaustive ones.	dynamic	
<code>-fmax-valid-substitutions={n}</code> (page 92)	<i>default:</i> 6. Set the maximum number of valid substitutions for typed holes to display in type error messages.	dynamic	<code>-fno-max-valid-substitutions</code>
<code>-fmax-worker-args={n}</code> (page 93)	<i>default:</i> 10. If a worker has that many arguments, none will be unpacked anymore.	dynamic	
<code>-fno-opt-coercion</code> (page 93)	Turn off the coercion optimiser	dynamic	
<code>-fno-pre-inlining</code> (page 93)	Turn off pre-inlining	dynamic	
<code>-fno-state-hack</code> (page 93)	Turn off the state hack whereby any lambda with a real-world state token as argument is considered to be single-entry. Hence OK to inline things inside it.	dynamic	
<code>-fomit-interface-pragmas</code> (page 93)	Don't generate interface pragmas. Implied by <code>-O0</code> (page 88) only.	dynamic	<code>-fno-omit-interface-pragmas</code>
<code>-fomit-yields</code> (page 93)	Omit heap checks when no allocation is being performed.	dynamic	<code>-fno-omit-yields</code>
Continued on next page			

Table 6.14 – continued from previous page

Flag	Description	Type	Reverse
<i>-foptimal-applicative-do</i> (page 250)	Use a slower but better algorithm for ApplicativeDo	dynamic	-fno-optimal-applicative-do
<i>-fpedantic-bottoms</i> (page 93)	Make GHC be more precise about its treatment of bottom (but see also <i>-fno-state-hack</i> (page 93)). In particular, GHC will not eta-expand through a case expression.	dynamic	-fno-pedantic-bottoms
<i>-fregs-graph</i> (page 94)	Use the graph colouring register allocator for register allocation in the native code generator. Implied by <i>-O2</i> (page 88).	dynamic	-fno-regs-graph
<i>-fregs-iterative</i> (page 94)	Use the iterative coalescing graph colouring register allocator in the native code generator.	dynamic	-fno-regs-iterative
<i>-fsimpl-tick-factor={n}</i> (page 94)	<i>default: 100.</i> Set the percentage factor for simplifier ticks.	dynamic	
<i>-fsimplifier-phases={n}</i> (page 94)	<i>default: 2.</i> Set the number of phases for the simplifier. Ignored with <i>-O0</i> (page 88).	dynamic	
<i>-fsolve-constant-dicts</i> (page 96)	When solving constraints, try to eagerly solve super classes using available dictionaries.	dynamic	-fno-solve-constant-dicts
<i>-fspec-constr</i> (page 94)	Turn on the SpecConstr transformation. Implied by <i>-O2</i> (page 88).	dynamic	-fno-spec-constr
<i>-fspec-constr-count={n}</i> (page 95)	<i>default: 3.*</i> 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
<i>-fspec-constr-keen</i> (page 95)	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
<i>-fspec-constr-threshold={n}</i> (page 95)	<i>default: 2000.</i> Set the size threshold for the SpecConstr transformation to {n}.	dynamic	-fno-spec-constr-threshold
Continued on next page			

Table 6.14 – continued from previous page

Flag	Description	Type	Reverse
<i>-fspecialise</i> (page 95)	Turn on specialisation of overloaded functions. Implied by <i>-O</i> (page 88).	dynamic	<i>-fno-specialise</i>
<i>-fspecialise-aggressively</i> (page 96)	Turn on specialisation of overloaded functions regardless of size, if unfolding is available	dynamic	<i>-fno-specialise-aggressively</i>
<i>-fstatic-argument-transformation</i> (page 96)	Turn on the static argument transformation.	dynamic	<i>-fno-static-argument-transformation</i>
<i>-fstg-cse</i> (page 90)	Enable common sub-expression elimination on the STG intermediate language	dynamic	<i>-fno-stg-cse</i>
<i>-fstrictness</i> (page 96)	Turn on strictness analysis. Implied by <i>-O</i> (page 88). Implies <i>-fworker-wrapper</i>	dynamic	<i>-fno-strictness</i>
<i>-fstrictness-before={n}</i> (page 97)	Run an additional strictness analysis before simplifier phase {n}	dynamic	
<i>-funbox-small-strict-fields</i> (page 97)	Flatten strict constructor fields with a pointer-sized representation. Implied by <i>-O</i> (page 88).	dynamic	<i>-fno-unbox-small-strict-fields</i>
<i>-funbox-strict-fields</i> (page 97)	Flatten strict constructor fields	dynamic	<i>-fno-unbox-strict-fields</i>
<i>-funfolding-creation-threshold={n}</i> (page 97)	<i>default: 750.</i> Tweak unfolding settings.	dynamic	
<i>-funfolding-dict-discount={n}</i> (page 98)	<i>default: 30.</i> Tweak unfolding settings.	dynamic	
<i>-funfolding-fun-discount={n}</i> (page 98)	<i>default: 60.</i> Tweak unfolding settings.	dynamic	
<i>-funfolding-keenness-factor={n}</i> (page 98)	<i>default: 1.5.</i> Tweak unfolding settings.	dynamic	
<i>-funfolding-use-threshold={n}</i> (page 98)	<i>default: 60.</i> Tweak unfolding settings.	dynamic	
<i>-fvectorisation-avoidance</i> (page 98)	Enable vectorisation avoidance. Always enabled by default.	dynamic	<i>-fno-vectorisation-avoidance</i>
<i>-fvectorise</i> (page 98)	Enable vectorisation of nested data parallelism	dynamic	<i>-fno-vectorise</i>

6.6.16 Profiling options

More details in *Profiling* (page 203)

Flag	Description	Type	Reverse
<i>-fno-prof-auto</i> (page 208)	Disables any previous <i>-fprof-auto</i> (page 207), <i>-fprof-auto-top</i> (page 207), or <i>-fprof-auto-exported</i> (page 207) options.	dynamic	<i>-fprof-auto</i> (page 207)
<i>-fno-prof-cafs</i> (page 208)	Disables any previous <i>-fprof-cafs</i> (page 207) option.	dynamic	<i>-fprof-cafs</i> (page 207)
<i>-fno-prof-count-entries</i> (page 208)	Do not collect entry counts	dynamic	<i>-fprof-count-entries</i>
<i>-fprof-auto</i> (page 207)	Auto-add SCC\ s to all bindings not marked INLINE	dynamic	<i>-fno-prof-auto</i> (page 208)
<i>-fprof-auto-calls</i> (page 207)	Auto-add SCC\ s to all call sites	dynamic	<i>-fno-prof-auto-calls</i>
<i>-fprof-auto-exported</i> (page 207)	Auto-add SCC\ s to all exported bindings not marked INLINE	dynamic	<i>-fno-prof-auto</i> (page 208)
<i>-fprof-auto-top</i> (page 207)	Auto-add SCC\ s to all top-level bindings not marked INLINE	dynamic	<i>-fno-prof-auto</i> (page 208)
<i>-fprof-cafs</i> (page 207)	Auto-add SCC\ s to all CAFs	dynamic	<i>-fno-prof-cafs</i> (page 208)
<i>-prof</i> (page 207)	Turn on profiling	dynamic	
<i>-ticky</i> (page 225)	<i>Turn on ticky-ticky profiling</i> (page 225)	dynamic	

6.6.17 Program coverage options

More details in *Observing Code Coverage* (page 220)

Flag	Description	Type	Reverse
<i>-fhpc</i> (page 221)	Turn on Haskell program coverage instrumentation	dynamic	

6.6.18 C pre-processor options

More details in *Options affecting the C pre-processor* (page 183)

Flag	Description	Type	Reverse
<i>-cpp</i> (page 183)	Run the C pre-processor on Haskell source files	dynamic	
<i>-D{symbol}[={value}]</i> (page 183)	Define a symbol in the C pre-processor	dynamic	<i>-U{symbol}</i> (page 183)

Continued on next page

Table 6.17 – continued from previous page

Flag	Description	Type	Reverse
<code>-I<dir></code> (page 183)	Add <dir> to the directory search list for <code>#include</code> files	dynamic	
<code>-U<symbol></code> (page 183)	Undefine a symbol in the C pre-processor	dynamic	

6.6.19 Code generation options

More details in *Options affecting code generation* (page 186)

Flag	Description	Type	Reverse
<code>-dynamic-too</code> (page 187)	Build dynamic object files <i>as well as</i> static object files during compilation	dynamic	
<code>-fasm</code> (page 186)	Use the <i>native code generator</i> (page 180)	dynamic	<code>-fllvm</code> (page 186)
<code>-fbyte-code</code> (page 186)	Generate byte-code	dynamic	
<code>-fllvm</code> (page 186)	Compile using the <i>LLVM code generator</i> (page 180)	dynamic	<code>-fasm</code> (page 186)
<code>-fno-code</code> (page 186)	Omit code generation	dynamic	
<code>-fobject-code</code> (page 186)	Generate object code	dynamic	
<code>-fPIC</code> (page 186)	Generate position-independent code (where available)	dynamic	
<code>-fPIE</code> (page 186)	Generate code for a position-independent executable (where available)	dynamic	
<code>-fwrite-interface</code> (page 186)	Always write interface files	dynamic	

6.6.20 Linking options

More details in *Options affecting linking* (page 187)

Flag	Description	Type	Reverse
<code>-c</code> (page 66)	Stop after generating object (.o) file	mode	
<code>-debug</code> (page 189)	Use the debugging runtime	dynamic	
Continued on next page			

Table 6.19 - continued from previous page

Flag	Description	Type	Reverse
<code>-dylib-install-name {path}</code> (page 191)	Set the install name (via <code>-install_name</code> 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	
<code>-dynamic</code> (page 188)	Build dynamically-linked object files and executables	dynamic	
<code>-dynload</code> (page 189)	Selects one of a number of modes for finding shared libraries at runtime.	dynamic	
<code>-eventlog</code> (page 190)	Enable runtime event tracing	dynamic	
<code>-fno-embed-manifest</code> (page 191)	Do not embed the manifest in the executable (Windows only)	dynamic	
<code>-fno-gen-manifest</code> (page 191)	Do not generate a manifest file (Windows only)	dynamic	
<code>-fno-shared-implib</code> (page 191)	Don't generate an import library for a DLL (Windows only)	dynamic	
<code>-framework {name}</code> (page 187)	On Darwin/OS X/iOS only, link in the framework {name}. This option corresponds to the <code>-framework</code> option for Apple's Linker.	dynamic	
<code>-framework-path {dir}</code> (page 188)	On Darwin/OS X/iOS only, add {dir} to the list of directories searched for frameworks. This option corresponds to the <code>-F</code> option for Apple's Linker.	dynamic	
<code>-fwhole-archive-hs-libs</code> (page 192)	When linking a binary executable, this inserts the flag <code>-Wl,--whole-archive</code> before any <code>-l</code> flags for Haskell libraries, and <code>-Wl,--no-whole-archive</code> afterwards	dynamic	
<code>-L {dir}</code> (page 188)	Add {dir} to the list of directories searched for libraries	dynamic	
<code>-l {lib}</code> (page 187)	Link in library {lib}	dynamic	
<code>-main-is {thing}</code> (page 189)	Set main module and function	dynamic	
<code>-no-hs-main</code> (page 189)	Don't assume this program contains main	dynamic	
<code>-no-rtsopts-suggestions</code> (page 191)	Don't print RTS suggestions about linking with <code>-rtsopts[=(none some all)]</code> (page 190).	dynamic	
<code>-package {name}</code> (page 187)	Expose package {pkg}	dynamic/ :set	
<code>-pie</code> (page 192)	Instruct the linker to produce a position-independent executable.	dynamic	

Continued on next page

Table 6.19 - continued from previous page

Flag	Description	Type	Reverse
<code>-rdynamic</code> (page 192)	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 <code>-optl -rdynamic</code> on Linux, and <code>-optl -export-all-symbols</code> on Windows.	dynamic	
<code>-rtsopts[=<i>{none some all}</i>]</code> (page 190)	Control whether the RTS behaviour can be tweaked via command-line flags and the <code>GHCRTS</code> environment variable. Using <code>none</code> means no RTS flags can be given; <code>some</code> means only a minimum of safe options can be given (the default), and <code>all</code> (or no argument at all) means that all RTS flags are permitted.	dynamic	
<code>-shared</code> (page 188)	Generate a shared library (as opposed to an executable)	dynamic	
<code>-split-objs</code> (page 188)	Split objects (for libraries)	dynamic	
<code>-split-sections</code> (page 188)	Split sections for link-time dead-code stripping	dynamic	
<code>-static</code> (page 188)	Use static Haskell libraries	dynamic	
<code>-staticlib</code> (page 188)	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 a single static library that can be linked against.	dynamic	
<code>-threaded</code> (page 189)	Use the threaded runtime	dynamic	
<code>-with-rtsopts=<i>{opts}</i></code> (page 190)	Set the default RTS options to <i>{opts}</i> .	dynamic	

6.6.21 Plugin options

More details in *Compiler Plugins* (page 475)

Flag	Description	Type	Reverse
<code>-fplugin-opt=<i>{module}</i>:<i>{args}</i></code> (page 475)	Give arguments to a plugin module; module must be specified with <code>-fplugin=<i>{module}</i></code> (page 475)	dynamic	
<code>-fplugin=<i>{module}</i></code> (page 475)	Load a plugin exported by a given module	dynamic	
<code>-hide-all-plugin-packages</code> (page 476)	Hide all packages for plugins by default	dynamic	
Continued on next page			

Table 6.20 – continued from previous page

Flag	Description	Type	Reverse
<code>-plugin-package <pkg></code> (page 476)	Expose <pkg> for plugins	dynamic	
<code>-plugin-package-id <pkg-id></code> (page 476)	Expose <pkg-id> for plugins	dynamic	

6.6.22 Replacing phases

More details in *Replacing the program for one or more phases* (page 181)

Flag	Description	Type	Reverse
<code>-pgma <cmd></code> (page 182)	Use <cmd> as the assembler	dynamic	
<code>-pgmc <cmd></code> (page 181)	Use <cmd> as the C compiler	dynamic	
<code>-pgmdl1 <cmd></code> (page 182)	Use <cmd> as the DLL generator	dynamic	
<code>-pgmF <cmd></code> (page 182)	Use <cmd> as the pre-processor (with -F only)	dynamic	
<code>-pgmi <cmd></code> (page 182)	Use <cmd> as the external interpreter command.	dynamic	
<code>-pgmL <cmd></code> (page 181)	Use <cmd> as the literate pre-processor	dynamic	
<code>-pgml <cmd></code> (page 182)	Use <cmd> as the linker	dynamic	
<code>-pgmlc <cmd></code> (page 181)	Use <cmd> as the LLVM compiler	dynamic	
<code>-pgmlibtool <cmd></code> (page 182)	Use <cmd> as the command for libtool (with -staticlib only).	dynamic	
<code>-pgmlo <cmd></code> (page 181)	Use <cmd> as the LLVM optimiser	dynamic	
<code>-pgmP <cmd></code> (page 181)	Use <cmd> as the C pre-processor (with -cpp only)	dynamic	
<code>-pgms <cmd></code> (page 182)	Use <cmd> as the splitter	dynamic	
<code>-pgmwindres <cmd></code> (page 182)	Use <cmd> as the program for embedding manifests on Windows.	dynamic	

6.6.23 Forcing options to particular phases

More details in *Forcing options to a particular phase* (page 182)

Flag	Description	Type	Reverse
<code>-opta <option></code> (page 182)	pass <option> to the assembler	dynamic	
<code>-optc <option></code> (page 182)	pass <option> to the C compiler	dynamic	

Continued on next page

Table 6.22 – continued from previous page

Flag	Description	Type	Reverse
<code>-optdll</code> <i>(option)</i> (page 182)	pass <i>(option)</i> to the DLL generator	dynamic	
<code>-optF</code> <i>(option)</i> (page 182)	pass <i>(option)</i> to the custom pre-processor	dynamic	
<code>-opti</code> <i>(option)</i> (page 183)	pass <i>(option)</i> to the interpreter sub-process.	dynamic	
<code>-optL</code> <i>(option)</i> (page 182)	pass <i>(option)</i> to the literate pre-processor	dynamic	
<code>-optl</code> <i>(option)</i> (page 182)	pass <i>(option)</i> to the linker	dynamic	
<code>-optlc</code> <i>(option)</i> (page 182)	pass <i>(option)</i> to the LLVM compiler	dynamic	
<code>-optlo</code> <i>(option)</i> (page 182)	pass <i>(option)</i> to the LLVM optimiser	dynamic	
<code>-optP</code> <i>(option)</i> (page 182)	pass <i>(option)</i> to cpp (with -cpp only)	dynamic	
<code>-optwindres</code> <i>(option)</i> (page 183)	pass <i>(option)</i> to windres.	dynamic	

6.6.24 Platform-specific options

More details in *Platform-specific Flags* (page 74)

Flag	Description	Type	Reverse
<code>-msse2</code> (page 74)	(x86 only) Use SSE2 for floating-point operations	dynamic	
<code>-msse4.2</code> (page 74)	(x86 only) Use SSE4.2 for floating-point operations	dynamic	

6.6.25 Compiler debugging options

More details in *Debugging the compiler* (page 195)

Flag	Description	Type	Reverse
<code>-dcmm-lint</code> (page 202)	C- pass sanity checking	dynamic	
<code>-dcore-lint</code> (page 202)	Turn on internal sanity checking	dynamic	
<code>-ddump-asm</code> (page 200)	Dump final assembly	dynamic	
<code>-ddump-asm-expanded</code> (page 200)	Dump the result of the synthetic instruction expansion pass.	dynamic	
<code>-ddump-asm-liveness</code> (page 200)	Dump assembly augmented with register liveness	dynamic	
<code>-ddump-asm-native</code> (page 200)	Dump initial assembly	dynamic	
<code>-ddump-asm-regalloc</code> (page 200)	Dump the result of register allocation	dynamic	

Continued on next page

Table 6.24 - continued from previous page

Flag	Description	Type	Reverse
<i>-ddump-asm-regalloc-stages</i> (page 200)	Dump the build/spill stages of the <i>-fregs-graph</i> (page 94) register allocator.	dynamic	
<i>-ddump-asm-stats</i> (page 200)	Dump statistics from the register allocator.	dynamic	
<i>-ddump-bcos</i> (page 200)	Dump interpreter byte code	dynamic	
<i>-ddump-cmm</i> (page 200)	Dump the final C- output	dynamic	
<i>-ddump-cmm-caf</i> (page 199)	Dump the results of the C- CAF analysis pass.	dynamic	
<i>-ddump-cmm-cbe</i> (page 199)	Dump the results of common block elimination	dynamic	
<i>-ddump-cmm-cfg</i> (page 199)	Dump the results of the C- control flow optimisation pass.	dynamic	
<i>-ddump-cmm-cps</i> (page 200)	Dump the results of the CPS pass	dynamic	
<i>-ddump-cmm-from-stg</i> (page 199)	Dump STG-to-C- output	dynamic	
<i>-ddump-cmm-info</i> (page 200)	Dump the results of the C- info table augmentation pass.	dynamic	
<i>-ddump-cmm-proc</i> (page 199)	Dump the results of proc-point analysis	dynamic	
<i>-ddump-cmm-procmap</i> (page 199)	Dump the results of the C- proc-point map pass.	dynamic	
<i>-ddump-cmm-raw</i> (page 199)	Dump raw C-	dynamic	
<i>-ddump-cmm-sink</i> (page 199)	Dump the results of the C- sinking pass.	dynamic	
<i>-ddump-cmm-sp</i> (page 199)	Dump the results of the C- stack layout pass.	dynamic	
<i>-ddump-cmm-split</i> (page 199)	Dump the results of the C- proc-point splitting pass.	dynamic	
<i>-ddump-cmm-switch</i> (page 199)	Dump the results of switch lowering passes	dynamic	
<i>-ddump-cmm-verbose</i> (page 199)	Show output from each C- pipeline pass	dynamic	
<i>-ddump-core-stats</i> (page 198)	Print a one-line summary of the size of the Core program at the end of the optimisation pipeline	dynamic	
<i>-ddump-cse</i> (page 198)	Dump CSE output	dynamic	
<i>-ddump-deriv</i> (page 197)	Dump deriving output	dynamic	
<i>-ddump-ds</i> (page 198)	Dump desugarer output	dynamic	
<i>-ddump-ec-trace</i> (page 197)	Trace exhaustiveness checker	dynamic	
<i>-ddump-foreign</i> (page 200)	Dump foreign export stubs	dynamic	

Continued on next page

Table 6.24 – continued from previous page

Flag	Description	Type	Reverse
<code>-ddump-if-trace</code> (page 197)	Trace interface files	dynamic	
<code>-ddump-inlinings</code> (page 198)	Dump inlining info	dynamic	
<code>-ddump-json</code> (page 196)	Dump error messages as JSON documents	dynamic	
<code>-ddump-llvm</code> (page 200)	Dump LLVM intermediate code.	dynamic	
<code>-ddump-occur-anal</code> (page 199)	Dump occurrence analysis output	dynamic	
<code>-ddump-opt-cmm</code> (page 200)	Dump the results of C- to C- optimising passes	dynamic	
<code>-ddump-parsed</code> (page 197)	Dump parse tree	dynamic	
<code>-ddump-parsed-ast</code> (page 197)	Dump parser output as a syntax tree	dynamic	
<code>-ddump-prep</code> (page 199)	Dump prepared core	dynamic	
<code>-ddump-rn</code> (page 197)	Dump renamer output	dynamic	
<code>-ddump-rn-ast</code> (page 197)	Dump renamer output as a syntax tree	dynamic	
<code>-ddump-rn-stats</code> (page 197)	Renamer stats	dynamic	
<code>-ddump-rn-trace</code> (page 197)	Trace renamer	dynamic	
<code>-ddump-rule-firings</code> (page 198)	Dump rule firing info	dynamic	
<code>-ddump-rule-rewrites</code> (page 198)	Dump detailed rule firing info	dynamic	
<code>-ddump-rules</code> (page 198)	Dump rewrite rules	dynamic	
<code>-ddump-simpl</code> (page 198)	Dump final simplifier output	dynamic	
<code>-ddump-simpl-iterations</code> (page 198)	Dump output from each simplifier iteration	dynamic	
<code>-ddump-simpl-stats</code> (page 198)	Dump simplifier stats	dynamic	
<code>-ddump-spec</code> (page 198)	Dump specialiser output	dynamic	
<code>-ddump-splices</code> (page 197)	Dump TH spliced expressions, and what they evaluate to	dynamic	
<code>-ddump-stg</code> (page 199)	Dump final STG	dynamic	
<code>-ddump-str-signatures</code> (page 198)	Dump strictness signatures	dynamic	
<code>-ddump-stranal</code> (page 198)	Dump strictness analyser output	dynamic	
<code>-ddump-tc</code> (page 197)	Dump typechecker output	dynamic	

Continued on next page

Table 6.24 – continued from previous page

Flag	Description	Type	Reverse
-ddump-tc-ast (page 197)	Dump typechecker output as a syntax tree	dynamic	
-ddump-tc-trace (page 197)	Trace typechecker	dynamic	
-ddump-timings (page 196)	Dump per-pass timing and allocation statistics	dynamic	
-ddump-to-file (page 196)	Dump to files instead of stdout	dynamic	
-ddump-types (page 197)	Dump type signatures	dynamic	
-ddump-vect (page 198)	Dump vectoriser input and output	dynamic	
-ddump-vt-trace (page 199)	Trace vectoriser	dynamic	
-ddump-worker-wrapper (page 198)	Dump worker-wrapper output	dynamic	
-dfaststring-stats (page 196)	Show statistics for fast string usage when finished	dynamic	
-dinitial-unique={s} (page 202)	Start UniqSupply allocation from {s}.	dynamic	
-dno-debug-output (page 201)	Suppress unsolicited debugging output	dynamic	- ddebug-output
-dppr-case-as-let (page 201)	Print single alternative case expressions as strict lets.	dynamic	
-dppr-cols={n} (page 201)	Set the width of debugging output. For example <code>-dppr-cols200</code>	dynamic	
-dppr-debug (page 196)	Turn on debug printing (more verbose)	dynamic	
-dppr-user-length (page 201)	Set the depth for printing expressions in error msgs	dynamic	
-dshow-passes (page 196)	Print out each pass name as it happens	dynamic	
-dstg-lint (page 202)	STG pass sanity checking	dynamic	
-dsuppress-all (page 201)	In core dumps, suppress everything (except for uniques) that is suppressible.	dynamic	
-dsuppress-coercions (page 201)	Suppress the printing of coercions in Core dumps to make them shorter	dynamic	
-dsuppress-idinfo (page 201)	Suppress extended information about identifiers where they are bound	dynamic	
-dsuppress-module-prefixes (page 201)	Suppress the printing of module qualification prefixes	dynamic	
-dsuppress-stg-free-vars (page 202)	Suppress the printing of closure free variable lists in STG output	dynamic	
-dsuppress-ticks (page 201)	Suppress “ticks” in the pretty-printer output.	dynamic	

Continued on next page

Table 6.24 – continued from previous page

Flag	Description	Type	Reverse
<code>-dsuppress-type-applications</code> (page 201)	Suppress type applications	dynamic	
<code>-dsuppress-type-signatures</code> (page 201)	Suppress type signatures	dynamic	
<code>-dsuppress-unfoldings</code> (page 201)	Suppress the printing of the stable unfolding of a variable at its binding site	dynamic	
<code>-dsuppress-uniques</code> (page 201)	Suppress the printing of uniques in debug output (easier to use diff)	dynamic	
<code>-dsuppress-var-kinds</code> (page 201)	Suppress the printing of variable kinds	dynamic	
<code>-dth-dec-file={file}</code> (page 197)	Show evaluated TH declarations in a .th.hs file	dynamic	
<code>-dunique-increment={i}</code> (page 202)	Set the increment for the generated Unique's to {i}.	dynamic	
<code>-dverbose-core2core</code> (page 198)	Show output from each core-to-core pass	dynamic	
<code>-dverbose-stg2stg</code> (page 199)	Show output from each STG-to-STG pass	dynamic	
<code>-falignment-sanitisation</code> (page 202)	Compile with alignment checks for all info table dereferences.	dynamic	
<code>-fcatch-bottoms</code> (page 202)	Insert error expressions after bottoming expressions; useful when debugging the compiler.	dynamic	
<code>-fllvm-fill-undef-with-garbage</code> (page 202)	Instruct LLVM to fill dead STG registers with garbage	dynamic	
<code>-g</code> (page 485), <code>-g{n}</code> (page 485)	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	

6.6.26 Miscellaneous compiler options

Flag	Description	Type	Reverse
<code>-fexternal-interpreter</code> (page 58)	Run interpreted code in a separate process	dynamic	
<code>-fglasgow-exts</code> (page 235)	Deprecated. Enable most language extensions; see <i>Language options</i> (page 233) for exactly which ones.	dynamic	<code>-fno-glasgow-exts</code>
<code>-ghcversion-file {path to ghcversion.h}</code> (page 74)	(GHC as a C compiler only) Use this ghcversion.h file	dynamic	
<code>-H {size}</code> (page 73)	Set the minimum size of the heap to {size}	dynamic	
<code>-j[{n}]</code> (page 68)	When compiling with <code>--make</code> (page 66), compile {n} modules in parallel.	dynamic	

6.7 Running a compiled program

To make an executable program, the GHC system compiles your code and then links it with a non-trivial runtime system (RTS), 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 131)) describes the various methods, and the following sections describe the RTS options themselves.

6.7.1 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 131))
- at compile-time, using `-with-rtsopts={opts}` (page 190) (*Setting RTS options at compile time* (page 132))
- with the environment variable `GHCRTS` (page 132) (*Setting RTS options with the GHCRTS environment variable* (page 132))
- by overriding “hooks” in the runtime system (*“Hooks” to change RTS behaviour* (page 132))

Setting RTS options on the command line

If you set the `-rtsopts[={none|some|all}]` (page 190) flag appropriately when linking (see *Options affecting linking* (page 187)), 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`.

As always, for RTS options that take (size)s: If the last character of (size) is a K or k, multiply by 1000; if an M or m, by 1,000,000; if a G or G, by 1,000,000,000. (And any wraparound in the counters is *your* fault!)

Giving a `+RTS -? RTS` option 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.

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 187)). 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"`.

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 132). 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 132) 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).

“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](#) (page 213)) is written through a custom [EventLogWriter](#) (page 133):

EventLogWriter

A sink of event-log data.

void **initEventLogWriter**(void)

Initializes your [EventLogWriter](#) (page 133). This is optional.

bool **writeEventLog**(void *eventlog, size_t eventlog_size)

Hands buffered event log data to your event log writer. Required for a custom [EventLogWriter](#) (page 133).

void **flushEventLog**(void)

Flush buffers (if any) of your custom [EventLogWriter](#) (page 133). This can be NULL.

void **stopEventLogWriter**(void)

Called when event logging is about to stop. This can be NULL.

6.7.2 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 208)).

--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.

-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”. 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 -xm200000000 -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 `-xm400000000`.

-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.

6.7.3 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.

-A {size}

Default 1MB

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 137), below).

Increasing the allocation area size may or may not give better performance (a bigger allocation area means worse cache behaviour but fewer garbage collections and less promotion).

With only 1 generation (e.g. `-G1`, see `-G {generations}` (page 136)) 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 135), below).

-AL {size}

Default `-A` (page 134) 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 134) 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 reserved 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 135) option multiplied by the size of the live data at the previous major collection, whichever is larger.

-n *(size)*

Default 4m with `-A16m` (page 134) or larger, otherwise 0.

[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 73) 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 138) 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 138).

-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 137)) than to increase `-F <factor>` (page 135).

The `-F <factor>` (page 135) setting will be automatically reduced by the garbage collector when the maximum heap size (the `-M <size>` (page 138) setting) is approaching.

-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 134) 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 137) 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 100) option, and additionally it might be beneficial to restrict parallel GC to the old generation with `-qg1`.

-qb <gen>

Default 1 for `-A` (page 134) < 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 it is 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 100) or the number of CPU cores, whichever is smaller.

Since 8.2.1

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 134) 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 and SMP versions of the RTS (see *-threaded* (page 189), *Options affecting linking* (page 187)), a major GC is automatically performed if the runtime has been idle (no Haskell computation has been running) for a period of time. The amount of idle time which must pass before a GC is performed is set by the `-I {seconds}` option. Specifying `-I0` disables the idle GC.

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.

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 138) 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 138)), 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.

-M*grace*={size}

Default 1M

If the program's heap exceeds the value set by `-M {size}` (page 138), 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. `-M`*grace*= 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 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 93).

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 485) 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.

6.7.4 RTS options to produce runtime statistics

-T**-t** [{file}]**-s** [{file}]**-S** [{file}]**--machine-readable**

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.

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
```

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 143) RTS option (*RTS options for profiling* (page 143)).

- 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))
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 bytes	Copied bytes	Live bytes	GC user	GC elap	TOT user	TOT elap	Page	Flts	
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.

6.7.5 RTS options for concurrency and parallelism

The RTS options related to concurrency are described in *Using Concurrent Haskell* (page 99), and those for parallelism in *RTS options for SMP parallelism* (page 100).

6.7.6 RTS options for profiling

Most profiling runtime options are only available when you compile your program for profiling (see *Compiler options for profiling* (page 207), and *RTS options for heap profiling* (page 212) 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 216)). 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 203)). Can be shortened to **-h** (page 143).

-L `<n>`

Default 25 characters

Sets the maximum length of the cost-centre names listed in the heap profile.

6.7.7 Tracing

When the program is linked with the **-eventlog** (page 190) option (*Options affecting linking* (page 187)), 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.
- **p** — parallel sparks (sampled). Enabled by default.

- **f** — parallel sparks (fully accurate). 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 by the header `EventLogFormat.h` that comes with GHC, and 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.

-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.

6.7.8 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 (major) 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 189) 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.

Debug messages will be sent to the binary event log file instead of stdout if the `-l` (page 213) option is added. This might be useful for reducing the overhead of debug tracing.

-r {file}

Produce “ticky-ticky” statistics at the end of the program run (only available if the program was linked with `-debug` (page 189)). The {file} business works just like on the `-S [{file}]` (page 140) RTS option, above.

For more information on ticky-ticky profiling, see *Using “ticky-ticky” profiling (for implementors)* (page 225).

-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 207)) 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” at garbage-collection time. (There's no particularly good reason to turn it off, except to ensure the accuracy of certain data collected regarding thunk entry counts.)

6.7.9 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 146) 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 unregistered", "NO")
,("Tables next to code", "YES")
]
```

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 189) option) and `rts_p` (profiling runtime, i.e. linked using the `-prof` (page 207) option). Other variants include `debug` (linked using `-debug` (page 189)), 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 platform**Target architecture****Target OS****Target vendor** These are the platform the program is compiled to run on.

Build platform**Build architecture****Build OS****Build 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 platform**Host architecture****Host OS****Host 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 181) 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.)

6.8 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.

6.8.1 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.

6.8.2 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 66) option instead (see *Other options related to interface files* (page 151)).

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 150) and `-hisuf <suffix>` (page 150) instead of `-odir <dir>` (page 149) and `-osuf <suffix>` (page 150) 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 148). 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 149) option, and the name of the interface file can be specified directly using the `-ohi <file>` (page 149) option.

6.8.3 The search path

In your program, you import a module `Foo` by saying `import Foo`. In `--make` (page 66) mode or `GHCi`, GHC will look for a source file for `Foo` and arrange to compile it first. Without `--make` (page 66), GHC will look for the interface file for `Foo`, which should have been created by an earlier compilation of `Foo`. GHC uses the same strategy in each of these cases for finding the appropriate file.

This strategy 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 66) mode or `GHCi`, or `<hisuf>` otherwise.

For example, suppose the search path contains directories `d1`, `d2`, and `d3`, and we are in `--make` (page 66) 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 163)) for details.

6.8.4 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).

-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`.

-ohi {file}

The interface output may be directed to another file `bar2/Wurple.iface` with the option `-ohi bar2/Wurple.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.

-hidir {dir}

Redirects all generated interface files into {dir}, instead of the default.

-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 464)). 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 149), `-hidir {dir}` (page 150), `-stubdir {dir}` (page 150) and `-dumpdir {dir}` (page 150).

-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.

-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 151)).

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.

-hcsuf {suffix}

Finally, the option `-hcsuf {suffix}` will change the `.hc` file suffix for compiler-generated intermediate C files.

6.8.5 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 181) (Note: `.hc` files are only generated by [unregisterised](#) (page 181) 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-llvm-file

-keep-llvm-files

Implies `-fllvm` (page 186)

Keep intermediate `.ll` files when doing `.hs-to-.o` compilations via *LLVM* (page 180) (Note: `.ll` files aren't generated when using the native code generator, you may need to use `-fllvm` (page 186) 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 151)). Running GHC with `-v` will show you what temporary files were generated along the way.

6.8.6 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 151) 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.

6.8.7 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 151) 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 150) 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 66).

6.8.8 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 source file's modification date is earlier than the .o file's date (i.e. the source hasn't changed since the file was last compiled), 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.

6.8.9 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 160)).
 - 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 `-ohidir` 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 264)), 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 369)), 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 is exactly as in Haskell, except that you may not put default method declarations. You can also omit all the superclasses and class methods entirely; but you must either omit them all or put them all in.
- 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 442).)

6.8.10 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 Text
  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 166) 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 `Sig-Name` 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 [Trac #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 [Trac #12717](#)

6.8.11 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
OBSJS = Main.o   Foo.o   Bar.o

.SUFFIXES : .o .hs .hi .lhs .hc .s

cool_pgm : $(OBSJS)
    rm -f $@
    $(HC) -o $@ $(HC_OPTS) $(OBSJS)

# 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 152)).

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 161)

6.8.12 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 -dep-suffix '' -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 152) 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 package².

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).

² This is a change in behaviour relative to 6.2 and earlier.

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 69).)

-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`. Note that you must provide at least one suffix; if you do not want a suffix then pass `-dep-suffix ''`.

--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.

6.8.13 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 [Relaxed rules for instance contexts](#) (page 321)), 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 orphan instance
 - 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 83), GHC will warn you if you are creating an orphan module. Like any warning, you can switch the warning off with `-Wno-orphans` (page 83), and `-Werror` (page 75) 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 66) `mode` (page 66). If there is a `[orphan module]` on the first line, GHC considers it an orphan module.

6.9 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 164).

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).

6.9.1 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 168).

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)
  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
  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 449)) 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 172)):

```
$ ghc-pkg field network exposed-modules
exposed-modules: Network.BSD,
                  Network.CGI,
                  Network.Socket,
                  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 166) 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 166) 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 167).

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 188)/ `-dynamic` (page 188).

In `--make` (page 66) mode and `--interactive` (page 66) mode (see [Modes of operation](#) (page 66)), the compiler normally determines which packages are required by the cur-

rent 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 165), 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 167).

-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 165) 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 165): 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 166) 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 166) 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 165) 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 449)).

-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 165) 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 449)).

-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 457) options. This option does not change the exposed/hidden status of a package, so it isn't equivalent to applying `-distrust (pkg)` (page 457) to all packages on the system. (see *Safe Haskell* (page 449)).

6.9.2 The main package

Every complete Haskell program must define `main` in module `Main` in package `main`. Omitting the `-this-unit-id (unit-id)` (page 166) 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
```

6.9.3 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 own³, 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.

6.9.4 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

³ it used to in GHC 6.4, but not since 6.6

between these modules was to use *Package-qualified imports* (page 261). However, since GHC 7.10, the `-package (pkg)` (page 165) 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.

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 256) extension.

6.9.5 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 172)).

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 `$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 172)).

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 169) 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 69) flag.

This stack structure means that the order of `-package-db (file)` (page 169) flags or `GHC_PACKAGE_PATH` (page 169) 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 169) 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 169) 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 169) to specify the position in the stack where the user's package database should be loaded.

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 168) 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=$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.

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.

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 166) and the fact that it is *-package-id {unit-id}* (page 166), not *-package {pkg}* (page 165). 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 `$HOME/.ghc/arch-os-version/environments/{name}`

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 170).
- File `$HOME/.ghc/arch-os-version/environments/name` if you pass the option *-package-env {name}*.

- File `{file}` if the environment variable `GHC_ENVIRONMENT` is set to `{file}`.
- File `$HOME/.ghc/arch-os-version/environments/name` if the environment variable `GHC_ENVIRONMENT` 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 `$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.

6.9.6 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:
  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.

6.9.7 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 169) environment variable (see [The GHC_PACKAGE_PATH environment variable](#) (page 169), and using `-package-db {file}` (page 169) 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 169) 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 177).

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 177) 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** Causes `ghc-pkg` to interpret arguments as installed package 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).
- package-key** Causes `ghc-pkg` to interpret arguments as unit IDs (e.g., an identifier like `I5BErHzy0m07EBNpKBEeUv`). Package keys are used to prefix symbol names GHC produces (e.g., `6VWy06pWzzJq9evDvK2d4w6_DataziByteStringziInternal_unsafePackLenChars_info`), so if you need to figure out what package a symbol belongs to, use `ghc-pkg` with this flag.

6.9.8 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 177).

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-objs* (page 188) then building the HSfoo.o is worthwhile because it saves time when loading the package into GHCi. Without *-split-objs* (page 188), 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 188)/*-dynamic* (page 188) flags work, see *Options affecting linking* (page 187).

```
ghc -shared 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 186) on how object files must be prepared for shared object linking.

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 164)). 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.

6.9.9 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-de7803fla8cd88d2161b29b083c94240
license: BSD3
copyright:
maintainer: libraries@haskell.org
stability:
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 re-naming), 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 module is 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. Note that we don't support building dynamic libraries of Haskell code on Unix 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.

6.10 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.

6.10.1 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.

6.10.2 LLVM Code Generator (`-fllvm`)

This is an alternative backend that uses the [LLVM](#) compiler to produce executable code. It generally produces code as 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 186) 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](#) available on your `PATH`.

Note: Note that this GHC release expects an LLVM version in the 5.0 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](#).

6.10.3 C Code Generator (-fvia-C)

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` flag.

The C code generator is only supported when GHC is built in unregistered 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 67)).

6.10.4 Unregistered compilation

The term “unregistered” really means “compile via vanilla C”, disabling some of the platform-specific tricks that GHC normally uses to make programs go faster. When compiling unregistered, GHC simply generates a C file which is compiled via `gcc`.

When GHC is built in unregistered mode only the LLVM and C code generators will be available. The native code generator won’t be. LLVM usually offers a substantial performance benefit over the C backend in unregistered mode.

Unregistered 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 unregistered compilation going is usually fairly small.

Unregistered 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 unregistered by calling `ghc --info` (see [--info](#) (page 67)).

6.11 Options related to a particular phase

6.11.1 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` only).
- pgmc** `{cmd}`
Use `{cmd}` as the C compiler.
- pgmlo** `{cmd}`
Use `{cmd}` as the LLVM optimiser.

- pgmlc** {cmd}
Use {cmd} as the LLVM compiler.
- pgms** {cmd}
Use {cmd} as the splitter.
- pgma** {cmd}
Use {cmd} as the assembler.
- pgml** {cmd}
Use {cmd} as the linker.
- pgmdll** {cmd}
Use {cmd} as the DLL generator.
- pgmF** {cmd}
Use {cmd} as the pre-processor (with -F only).
- 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 187).
- pgmlibtool** {cmd}
Use {cmd} as the libtool command (when using `-staticlib` only).
- pgmi** {cmd}
Use {cmd} as the external interpreter command (see: *Running the interpreter in a separate process* (page 58)). Default: `ghc-iserv-prof` if `-prof` is enabled, `ghc-iserv-dyn` if `-dynamic` is enabled, or `ghc-iserv` otherwise.

6.11.2 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` is also on).
- optF** {option}
Pass {option} to the custom pre-processor (see *Options affecting a Haskell pre-processor* (page 185)).
- optc** {option}
Pass {option} to the C compiler.
- optlo** {option}
Pass {option} to the LLVM optimiser.
- optlc** {option}
Pass {option} to the LLVM compiler.
- opta** {option}
Pass {option} to the assembler.
- optl** {option}
Pass {option} to the linker.
- optdll** {option}
Pass {option} to the DLL generator.

-optwindres {option}

Pass {option} to windres when embedding manifests on Windows. See -fno-embed-manifest in *Options affecting linking* (page 187).

-opti {option}

Pass {option} to the interpreter sub-process (see *Running the interpreter in a separate process* (page 58)). 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 -Ewurbble option to the assembler, you would tell the driver -opta-Ewurbble (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 *Running a compiled program* (page 131)).

6.11.3 Options affecting the C pre-processor

CPP

Since 6.8.1

-cpp

The C pre-processor **cpp** is run over your Haskell code only if the -cpp option -cpp option is 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 183) does *not* affect -D macros passed to the C compiler when compiling an unregistered build! In this case use the -optc-Dfoo hack... (see *Forcing options to a particular phase* (page 182)).

-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).

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 <xy> (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 6).

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_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.:

```
#ifdef 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 `-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.

CPP and string gaps

A small word of warning: `-cpp` (page 183) is not friendly to “string gaps”. In other words, strings such as the following:

```
strmod = "\
\ p \
\"
```

don't work with `-cpp` (page 183); `/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.

6.11.4 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 reaping the benefits of GHC's recompilation checker.

The pre-processor is run just before the Haskell compiler properly 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 182) 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 182) 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 \"\$2\" #-}" ; iconv -f ll -t utf-8 $2 ) > $3
```

and pass `-F -pgmF convert.sh` to GHC. The `-f ll` 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 \"\$2\" #-}"` just makes sure that your error positions are reported as in the original source file.

6.11.5 Options affecting code generation

-fasm

Use GHC's *native code generator* (page 180) rather than compiling via LLVM. `-fasm` is the default.

-fllvm

Compile via *LLVM* (page 180) 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 in the 5.0 release series.

-fno-code

Omit code generation (and all later phases) altogether. This is useful if you're only interested in type checking code.

-fwrite-interface

Always write interface files. GHC will normally write interface files automatically, but this flag is useful with `-fno-code` (page 186), which normally suppresses generation of interface files. This is useful if you want to type check over multiple runs of GHC without compiling dependencies.

-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 bytecode.

-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 186).

-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.

-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 192).

-dynamic

When generating code, assume that entities imported from a different package will be dynamically linked. This can reduce code size tremendously, but may slow-down cross-package calls of non-inlined functions. There can be some complications combining *-shared* (page 188) with this flag relating to linking in the RTS under Linux. See [Trac #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.

6.11.6 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 *liblib.a* or *liblib.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 *-nohs-main*. See also [Using your own main\(\)](#) (page 465).

-c

Omits the link step. This option can be used with *--make* (page 66) 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 163)), 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

Link all passed files into a static library suitable for linking. To control the name, use the `-o {file}` (page 149) option as usual. The default name is `lib.a`.

-L {dir}

Where to find user-supplied libraries... Prepend the directory `{dir}` to the library directories path.

-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).

-split-objs

Tell the linker to split the single object file that would normally be generated into multiple object files, one per top-level Haskell function or type in the module. This only makes sense for libraries, where it means that executables linked against the library are smaller as they only link against the object files that they need. However, assembling all the sections separately is expensive, so this is slower than compiling normally. Additionally, the size of the library itself (the `.a` file) can be a factor of 2 to 2.5 larger. We use this feature for building GHC's libraries.

-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. The effect is similar to `-split-objs` (page 188), but somewhat more efficient - the generated library files are about 30% smaller than with `-split-objs` (page 188).

-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`). See [Using shared libraries](#) (page 192) 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 188) and `-static` (page 188) control whether the resulting shared object links statically or dynamically to Haskell package libraries given as `-package {pkg}` (page 165) 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 186), see [Options affecting code generation](#) (page 186)

When creating shared objects for Haskell packages, the shared object must be named properly, so that GHC recognizes the shared object when linked against this package. See shared object name mangling.

-dynload

This flag selects one of a number of modes for finding shared libraries at runtime. See [Finding shared libraries at runtime](#) (page 194) for a description of each mode.

-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 152) 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 465).

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 69) 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 66) 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}]` (page 190) and `-with-rtsopts={opts}` (page 190) have no effect when used with `-no-hs-main` (page 189), because they are implemented by changing the definition of `main` that GHC generates. See [Using your own main\(\)](#) (page 465) for how to get the effect of `-rtsopts[={none|some|all}]` (page 190) and `-with-rtsopts={opts}` (page 190) 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 100) RTS option to be used, which allows threads to run in parallel on a multiprocessor SMP or multicore machine. See [Using SMP parallelism](#) (page 99).
- 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 468).

-eventlog

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 143) for more information.

`-eventlog` (page 190) can be used with `-threaded` (page 189). It is implied by `-debug` (page 189).

-rtsopts[=(none|some|all)]

Default all

This option affects the processing of RTS control options given either on the command line or via the [GHCRTS](#) (page 132) environment variable. There are three 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 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 189); see [Using your own main\(\)](#) (page 465) for details.

-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.)

Note that -with-rtsopts has no effect when used with -no-hs-main; see [Using your own main\(\)](#) (page 465) for details.

-no-rtsopts-suggestions

This option disables RTS suggestions about linking with `-rtsopts[={none|some|all}]` (page 190) 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 191) also implies `-fno-embed-manifest` (page 191), 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 69) flag. A GHC installation comes with its own copy of windres for this reason.

See also `-pgmwindres {cmd}` (page 182) ([Replacing the program for one or more phases](#) (page 181)) and `-optwindres {option}` (page 183) ([Forcing options to a particular phase](#) (page 182)).

-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 188) 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 191) 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 186).

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 188) 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 192) flag to ensure that that symbols are not dropped from your PIE objects.

6.12 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 497)), 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.

6.12.1 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 188) 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).

6.12.2 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 188), 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 175). 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 188), `-fPIC` (page 186) and `-shared` (page 188).

6.12.3 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 467). 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 188), `-fPIC` (page 186) and `-shared` (page 188) flags:

```
ghc --make -dynamic -shared -fPIC Foo.hs -o libfoo.so
```

As before, the `-dynamic` (page 188) flag specifies that this library links against the shared library versions of the rts and base package. The `-fPIC` (page 186) flag is required for all code that will end up in a shared library. The `-shared` (page 188) 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 Foo.o -o libfoo.so
```

In principle you can use `-shared` (page 188) without `-dynamic` (page 188) 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 186) 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.

6.12.4 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.

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 192) flag during linking.

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 191) 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.

6.13 Debugging the compiler

HACKER TERRITORY. HACKER TERRITORY. (You were warned.)

Dump flags

- *Debugging the compiler* (page 195)
 - *Dumping out compiler intermediate structures* (page 196)
 - * *Front-end* (page 197)
 - * *Type-checking and renaming* (page 197)
 - * *Core representation and simplification* (page 198)
 - * *STG representation* (page 199)
 - * *C- representation* (page 199)
 - * *LLVM code generator* (page 200)
 - * *Native code generator* (page 200)
 - * *Miscellaneous backend dumps* (page 200)
 - *Formatting dumps* (page 201)
 - *Suppressing unwanted information* (page 201)
 - *Checking for consistency* (page 202)
 - *Checking for determinism* (page 202)

6.13.1 Dumping out compiler intermediate structures

-ddump-to-file

Causes the output from all of the flags listed below to be dumped to a file. The file name depends upon the output produced; for instance, output from *-ddump-simpl* (page 198) will end up in *module.dump-simpl*.

-ddump-json

Dump error messages as JSON documents. This is intended to be consumed by external tooling. A good way to use it is in conjunction with *-ddump-to-file* (page 196).

-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.

-dfaststring-stats

Show statistics on the usage of fast strings by the compiler.

-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 forall’s, 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.

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 196).

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

-ddump-if-trace

Make the interface loader be *real* chatty about what it is up to.

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-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

-ddump-tc-ast

Dump typechecker output as a syntax tree

-ddump-splices

Dump Template Haskell expressions that we splice in, and what Haskell code the expression evaluates to.

-dth-dec-file=<file>

Dump expansions of all top-level Template Haskell splices into <file>.

-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 196) dumps a type signature for all the imported and system-defined things as well; useful for debugging the compiler.

-ddump-deriv

Dump derived instances

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-core-stats

Print a one-line summary of the size of the Core program at the end of the optimisation pipeline.

-ddump-ds

Dump desugarer output

-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). This outputs even more information than `-ddump-simpl-phases`.

-ddump-simpl-stats

Dump statistics about how many of each kind of transformation took place. If you add `-dppr-debug` you get more detailed information.

-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 specialisation pass

-ddump-rules

Dumps all rewrite rules specified in this module; see [Controlling what's going on in rewrite rules](#) (page 438).

-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

-ddump-vect

Dumps the output of the vectoriser.

-ddump-simpl

Dump simplifier output (Core-to-Core passes)

-ddump-inlinings

Dumps inlining info from the simplifier. Note that if used in conjunction with [-dverbose-core2core](#) (page 198) the compiler will also dump the inlinings that it considers but passes up, along with its rationale.

-ddump-stranal

Dump strictness analyser output

-ddump-str-signatures

Dump strictness signatures

-ddump-cse

Dump common subexpression elimination (CSE) pass output

-ddump-worker-wrapper

Dump worker/wrapper split output

-ddump-occur-anal

Dump “occurrence analysis” output

-ddump-vt-trace

Make the vectoriser be *real* chatty about what it is up to.

-ddump-prep

Dump output of Core preparation pass

STG representation

These flags dump various phases of GHC's STG pipeline.

-ddump-stg

Dump output of STG-to-STG passes

-dverbose-stg2stg

Show the output of the intermediate STG-to-STG pass. (*lots* of output!)

C- representation

These flags dump various phases of GHC's C- pipeline.

-ddump-cmm-verbose

Dump output from all C- pipeline stages. In case of .cmm compilation this also dumps the result of file parsing.

-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-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

LLVM code generator

-ddump-llvm

Implies *-fllvm* (page 186)

LLVM code from the *LLVM code generator* (page 180)

Native code generator

These flags dump various stages of the *native code generator's* (page 180) pipeline, which starts with C- and produces native assembler.

-ddump-opt-cmm

Dump the results of C- to C- optimising passes performed by the NCG.

-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 94) register allocator.

-ddump-asm-stats

Dump statistics from the register allocator.

-ddump-asm-expanded

Dump the result of the synthetic instruction expansion pass.

-ddump-asm

Dump assembly language produced by the

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-foreign

Dump foreign export stubs.

6.13.2 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.

-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.

6.13.3 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 201)

-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-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-var-kinds

Suppress the printing of variable kinds

-dsuppress-stg-free-vars

Suppress the printing of closure free variable lists in STG output

6.13.4 Checking for consistency

-dcore-lint

Turn on heavyweight intra-pass sanity-checking within GHC, at Core level. (It checks GHC's sanity, not yours.)

-dstg-lint

Ditto for STG level. (note: currently doesn't work).

-dcmm-lint

Ditto for C- 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 [Trac #11487](#)).

-falignment-sanitisation

Compile with alignment checks for all info table dereferences. This can be useful when finding pointer tagging issues.

-fcatch-bottoms

Instructs the simplifier to emit error expressions in the continuation of empty case analyses (which should bottom and consequently not return). This is helpful when debugging demand analysis bugs which can sometimes manifest as segmentation faults.

6.13.5 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 202) 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

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?”.

Profiling a program is a three-step process:

1. Re-compile your program for profiling with the `-prof` (page 207) option, and probably one of the options for adding automatic annotations: `-fprof-auto` (page 207) is the most common ¹.

If you are using external packages with **cabal**, you may need to reinstall these packages with profiling support; typically this is done with `cabal install -p package --reinstall`.

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 208)), 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.

7.1 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:

¹ `-fprof-auto` (page 207) was known as `-auto-all` prior to GHC 7.4.1.

```
$ ghc -prof -fprof-auto -rtsopts Main.hs
$ ./Main +RTS -p
121393
$
```

When a GHC-compiled program is run with the `-p` (page 208) 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

COST CENTRE MODULE                                no.      entries  individual
                                                %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 211)).

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:

```
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 208) 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 caller².

7.1.1 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.

The syntax of a cost centre annotation for expressions is

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.

```
{-# 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. (SCC stands for "Set Cost Centre"). The double quotes can be omitted if name is a Haskell identifier, 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

7.1.2 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 run-time.

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 207). This is especially useful when compiling with `-ffull-laziness` (page 91) (as is default with `-O` (page 88) 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 198)) in order to determine what these CAFs correspond to.

7.2 Compiler options for profiling

-prof

To make use of the profiling system *all* modules must be compiled and linked with the `-prof` (page 207) option. Any SCC annotations you’ve put in your source will spring to life.

Without a `-prof` (page 207) option, your SCCs are ignored; so you can compile SCC-laden code without changing it.

There are a few other profiling-related compilation options. Use them *in addition to* `-prof` (page 207). These do not have to be used consistently for all modules in a program.

-fprof-auto

All bindings not marked `INLINE`, whether exported or not, top level or nested, will be given automatic SCC annotations. Functions marked `INLINE` must be given a cost centre manually.

-fprof-auto-top

GHC will automatically add SCC annotations for all top-level bindings not marked `INLINE`. If you want a cost centre on an `INLINE` function, you have to add it manually.

-fprof-auto-exported

GHC will automatically add SCC annotations for all exported functions not marked `INLINE`. If you want a cost centre on an `INLINE` 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 145) RTS flag (*RTS options for hackers, debuggers, and over-interested souls* (page 144)) for more details.

-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...

-fno-prof-auto

Disables any previous *-fprof-auto* (page 207), *-fprof-auto-top* (page 207), or *-fprof-auto-exported* (page 207) options.

-fno-prof-cafs

Disables any previous *-fprof-cafs* (page 207) option.

-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).

7.3 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 208) 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 208) flag.

The *-P* (page 208) option produces a more detailed report containing the actual time and allocation data as well. (Not used much.)

The *-pa* (page 208) option produces the most detailed report containing all cost centres in addition to the actual time and allocation data.

-pj

The *-pj* (page 208) option produces a time/allocation profile report in JSON format written into the file <program>.prof.

-po <stem>

The *-po <stem>* (page 208) option overrides the stem used to form the output file paths for the cost-centre profiler (see *-p* (page 208) and *-pj* (page 208) flags above) and heap profiler (see *-h* (page 143)).

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 0.02

Sets the interval that the RTS clock ticks at, which is also the sampling interval of the time and allocation profile. The default is 0.02 seconds. 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 99)) and the heap profiling timer *RTS options for heap profiling*

(page 212). Also, the time profiler uses the RTS timer signal directly to record time profiling samples.

Normally, setting the `-V {secs}` (page 208) 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 99) or `-i {secs}` (page 214) options. However, setting `-V {secs}` (page 208) 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 hackers, debuggers, and over-interested souls* (page 144).

7.3.1 JSON profile format

When invoked with the `-pj` (page 208) flag the runtime will emit the cost-centre profile in a machine-readable JSON format. 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 100) 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",
      "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": []
  }
]
}
}

```

7.4 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.

To generate a heap profile from your program:

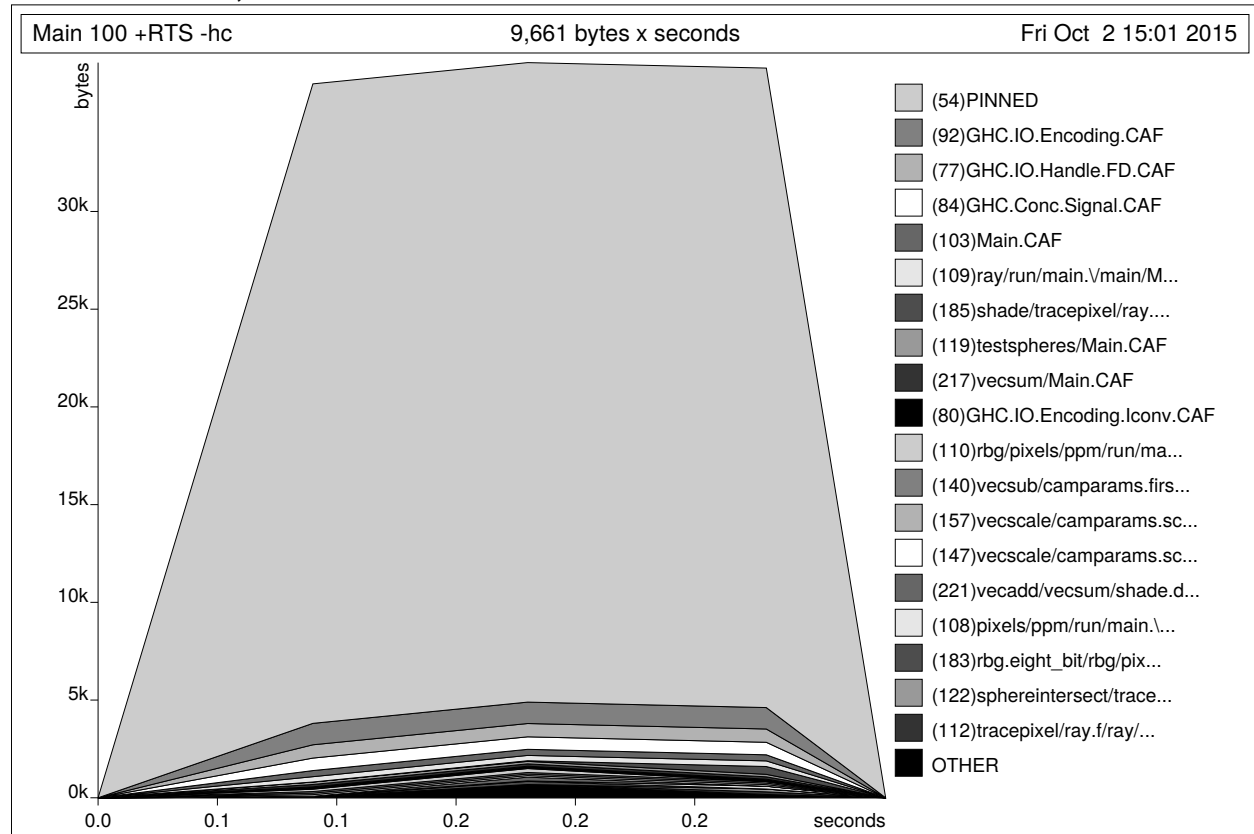
1. Compile the program for profiling (*Compiler options for profiling* (page 207)).
2. Run it with one of the heap profiling options described below (eg. `-h` (page 143) for a basic producer profile). This generates the file `prog.hp`.

If the *event log* (page 143) is enabled (with the `-l` (page 213) runtime system flag) heap samples will additionally be emitted to the GHC event log (see *Heap profiler event log output* (page 509) for details about event format).

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 216).

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,



You might also want to take a look at [hp2any](#), a more advanced suite of tools (not distributed with GHC) for displaying heap profiles.

7.4.1 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.

-hc

-h

Requires :ghc-flag: '-prof'. Breaks down the graph by the cost-centre stack which produced the data.

-hm

Requires :ghc-flag: '-prof'. Break down the live heap by the module containing the code which produced the data.

-hd

Requires :ghc-flag: '-prof'. 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 :ghc-flag:'-prof'. 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.

-hr

Requires :ghc-flag:'-prof'. Break down the graph by retainer set. Retainer profiling is described in more detail below ([Retainer Profiling](#) (page 214)).

-hb

Requires :ghc-flag:'-prof'. Break down the graph by biography. Biographical profiling is described in more detail below ([Biographical Profiling](#) (page 215)).

-l

Noindex

Emit profile samples to the [GHC event log](#) (page 143). 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 509) 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}

Restrict the profile to closures produced by cost-centre stacks with one of the specified cost centres at the top.

-hC {name}

Restrict the profile to closures produced by cost-centre stacks with one of the specified cost centres anywhere in the stack.

-hm {module}

Restrict the profile to closures produced by the specified modules.

-hd {desc}

Restrict the profile to closures with the specified description strings.

-hy {type}

Restrict the profile to closures with the specified types.

-hr {cc}

Restrict the profile to closures with retainer sets containing cost-centre stacks with one of the specified cost centres at the top.

-hb {bio}

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 213) 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 213) and `-hb` (page 213) 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 208) for changing that.

-xt

Include the memory occupied by threads in a heap profile. Each thread takes up a small area for its thread state in addition to the space allocated for its stack (stacks normally start small and then grow as necessary).

This includes the main thread, so using `-xt` (page 214) is a good way to see how much stack space the program is using.

Memory occupied by threads and their stacks is labelled as “TSO” and “STACK” respectively when displaying the profile by closure description or type description.

-L *{num}*

Sets the maximum length of a cost-centre stack name in a heap profile. Defaults to 25.

7.4.2 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 214) RTS option:

-R *{size}*

Restrict the number of elements in a retainer set to *{size}* (default 8).

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 Bs 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.

7.4.3 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.

7.4.4 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 135) 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 135) option), this is reduced to $2L$, and can further be reduced by tweaking the `-F {factor}` (page 135) option. Also add the size of the allocation area (see `-A {size}` (page 134)).
- The stack isn't counted in the heap profile by default. See the RTS `-xt` (page 214) option.
- 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.

7.5 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.

7.5.1 Manipulating the hp file

(Notes kindly offered by Jan-Willem Maessen.)

The `F00.hp` file produced when you ask for the heap profile of a program `F00` is a text file with a particularly simple structure. Here's a representative example, with much of the actual data

omitted:

```
JOB "F00 -hC"
DATE "Thu Dec 26 18:17 2002"
SAMPLE_UNIT "seconds"
VALUE_UNIT "bytes"
BEGIN_SAMPLE 0.00
END_SAMPLE 0.00
BEGIN_SAMPLE 15.07
... sample data ...
END_SAMPLE 15.07
BEGIN_SAMPLE 30.23
... sample data ...
END_SAMPLE 30.23
... etc.
BEGIN_SAMPLE 11695.47
END_SAMPLE 11695.47
```

The first four lines (JOB, DATE, SAMPLE_UNIT, VALUE_UNIT) form a header. Each block of lines starting with BEGIN_SAMPLE and ending with END_SAMPLE forms a single sample (you can think of this as a vertical slice of your heap profile). The hp2ps utility should accept any input with a properly-formatted header followed by a series of *complete* samples.

7.5.2 Zooming in on regions of your profile

You can look at particular regions of your profile simply by loading a copy of the .hp file into a text editor and deleting the unwanted samples. The resulting .hp file can be run through hp2ps and viewed or printed.

7.5.3 Viewing the heap profile of a running program

The .hp file is generated incrementally as your program runs. In principle, running **hp2ps** on the incomplete file should produce a snapshot of your program's heap usage. However, the last sample in the file may be incomplete, causing **hp2ps** to fail. If you are using a machine with UNIX utilities installed, it's not too hard to work around this problem (though the resulting command line looks rather Byzantine):

```
head -`fgrep -n END_SAMPLE F00.hp | tail -1 | cut -d : -f 1` F00.hp \
| hp2ps > F00.ps
```

The command `fgrep -n END_SAMPLE F00.hp` finds the end of every complete sample in F00.hp, and labels each sample with its ending line number. We then select the line number of the last complete sample using **tail** and **cut**. This is used as a parameter to **head**; the result is as if we deleted the final incomplete sample from F00.hp. This results in a properly-formatted .hp file which we feed directly to **hp2ps**.

7.5.4 Viewing a heap profile in real time

The **gv** and **ghostview** programs have a “watch file” option can be used to view an up-to-date heap profile of your program as it runs. Simply generate an incremental heap profile as described in the previous section. Run **gv** on your profile:


```
gv -watch -orientation=seascape F00.ps
```

If you forget the `-watch` flag you can still select “Watch file” from the “State” menu. Now each time you generate a new profile `F00.ps` the view will update automatically.

This can all be encapsulated in a little script:

```
#!/bin/sh
head -`fgrep -n END_SAMPLE F00.hp | tail -1 | cut -d : -f 1` F00.hp \
| hp2ps > F00.ps
gv -watch -orientation=seascape F00.ps &
while [ 1 ] ; do
  sleep 10 # We generate a new profile every 10 seconds.
  head -`fgrep -n END_SAMPLE F00.hp | tail -1 | cut -d : -f 1` F00.hp \
  | hp2ps > F00.ps
done
```

Occasionally `gv` will choke as it tries to read an incomplete copy of `F00.ps` (because `hp2ps` is still running as an update occurs). A slightly more complicated script works around this problem, by using the fact that sending a `SIGHUP` to `gv` will cause it to re-read its input file:

```
#!/bin/sh
head -`fgrep -n END_SAMPLE F00.hp | tail -1 | cut -d : -f 1` F00.hp \
| hp2ps > F00.ps
gv F00.ps &
gvpsnum=$!
while [ 1 ] ; do
  sleep 10
  head -`fgrep -n END_SAMPLE F00.hp | tail -1 | cut -d : -f 1` F00.hp \
  | hp2ps > F00.ps
  kill -HUP $gvpsnum
done
```

7.6 Profiling Parallel and Concurrent Programs

Combining `-threaded` (page 189) and `-prof` (page 207) is perfectly fine, and indeed it is possible to profile a program running on multiple processors with the RTS `-N <x>` (page 100) 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 208) 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.

7.7 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.

7.7.1 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.

```
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
  number <- readLn
```

```
putStrLn (showRecip number)
main
```

HPC instrumentation is enabled with the `-fhpc` (page 221) 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 221).

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`.

7.7.2 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 221). When the program is run, coverage data will only be generated for those modules that were compiled with `-fhpc` (page 221), and the `hpc` tool will only show information about those modules.

7.7.3 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.

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
```

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 than code gaps
  --destdir=DIR                 path to write output to
```

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)
```

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)
```

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)
```

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
```

7.7.4 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 221); the name of the `.tix` file will also change if you rename the binary. HPC does not work with GHCi.

7.8 Using “ticky-ticky” profiling (for implementors)

-ticky

Enable ticky-ticky profiling.

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.

ADVICE ON: SOONER, FASTER, SMALLER, THRIFTIER

Please advise us of other “helpful hints” that should go here!

8.1 Sooner: producing a program more quickly

Don’t use `-O` (page 88) or (especially) `-O2` (page 88): 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 88)!

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 137)) 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 134)).

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.

8.2 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 203). 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 180) can sometimes do a far better job at producing fast code than the [native code generator](#) (page 180). 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 182) and `-optlc {option}` (page 182) 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 82) 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 426) and [SPECIALIZE instance pragma](#) (page 429).

“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 66) option (see [Other options related to interface files](#) (page 151)).

```
$ ghc --show-iface Foo.hi | egrep '^[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 cases instead of lets to get stricter code (a good thing):

```
f (Wibble x y) # beautiful but slow
  = let
      (a1, b1, c1) = unpackFoo x
      (a2, b2, c2) = unpackFoo y
    in ...

f (Wibble x y) # ugly, and proud of it
  = case (unpackFoo x) of { (a1, b1, c1) ->
      case (unpackFoo y) of { (a2, b2, c2) ->
          ...
      }
  }
```

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 198); 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 `INLINEd` (esp. monads): Placing `INLINE` pragmas on certain functions that are used a lot can have a dramatic effect. See [`INLINE` pragma](#) (page 423).

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 198) (don't forget the `-O` (page 88)).

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., `eqInt#`) 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 97) (see `-f*`: [platform-independent flags](#) (page 88)), 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 97) is only really a good idea in conjunction with `-O` (page 88), because otherwise the extra packing and unpacking won't be optimised away. In fact, it is possible that `-funbox-strict-fields` (page 97) may worsen performance even with `-O` (page 88), 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 237) for some information about using unboxed types.

Before resorting to explicit unboxed types, try using strict constructor fields and `-funbox-strict-fields` (page 97) 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 461) 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 140) 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)` or `-A(size)` RTS options (see [RTS options to control the garbage collector](#) (page 134)). As a rule of thumb, try setting `-H (size)` to the amount of memory you're willing to let your process consume, or perhaps try passing `-H (size)` (page 73) 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.

8.3 Smaller: producing a program that is smaller

Decrease the “go-for-it” threshold for unfolding smallish expressions. Give a *-funfolding-use-threshold=0* (page 98) 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.

8.4 Thriftier: producing a program that gobbles less heap space

“I think I have a space leak...”

Re-run your program with `+RTS -S`, and remove all doubt! (You’ll see the heap usage get bigger and bigger...) (Hmmm... this might be even easier with the *-G1* (page 136) RTS option; so... `./a.out +RTS -S -G1`)

Once again, the profiling facilities (*Profiling* (page 203)) 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).

GHC LANGUAGE FEATURES

As with all known Haskell systems, GHC implements some extensions to the standard Haskell language. They can all be enabled or disabled by command line flags or language pragmas. By default GHC understands the most recent Haskell version it supports, plus a handful of extensions.

Some of the Glasgow 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!

Before you get too carried away working at the lowest level (e.g., sloshing `MutableByteArray#s` around your program), you may wish to check if there are libraries that provide a “Haskellised veneer” over the features you want. The separate libraries documentation describes all the libraries that come with GHC.

9.1 Language options

The language extensions control what variation of the language are permitted.

Language options can be controlled in two ways:

- Every language option can be switched on by a command-line flag “-X...” (e.g. `-XTemplateHaskell`), and switched off by the flag “-XNo...”; (e.g. `-XNoTemplateHaskell`).
- Language options recognised by Cabal can also be enabled using the `LANGUAGE` pragma, thus `{-# LANGUAGE TemplateHaskell #-}` (see [LANGUAGE pragma](#) (page 420)).

GHC supports these language options:

Extension	Description
AllowAmbiguousTypes (page 368)	Allow the user to write ambiguous types, and the type inference engine to infer them.
ApplicativeDo (page 248)	Enable Applicative do-notation desugaring
Arrows (page 404)	Enable arrow notation extension
BangPatterns (page 411)	Enable bang patterns.
BinaryLiterals (page 242)	Enable support for binary literals.
CApiFFI (page 463)	Enable the CAPI calling convention.
ConstrainedClassMethods (page 313)	Enable constrained class methods.
ConstraintKinds (page 366)	Enable a kind of constraints.

Extension	Description
CPP (page 183)	Enable the C preprocessor.
DataKinds (page 349)	Enable datatype promotion.
DatatypeContexts (page 264)	Allow contexts on data types.
DefaultSignatures (page 314)	Enable default signatures.
DeriveAnyClass (page 302)	Enable deriving for any class.
DeriveDataTypeable (page 295)	Enable deriving for the Data class. Implied by AutoDeriveTypeable .
DeriveFoldable (page 292)	Enable deriving for the Foldable class. Implied by DeriveTraversable .
DeriveFunctor (page 289)	Enable deriving for the Functor class. Implied by DeriveTraversable .
DeriveGeneric (page 440)	Enable deriving for the Generic class.
DeriveLift (page 296)	Enable deriving for the Lift class.
DeriveTraversable (page 294)	Enable deriving for the Traversable class. Implies DeriveFunctor .
DerivingStrategies (page 304)	Enables deriving strategies.
DisambiguateRecordFields (page 277)	Enable record field disambiguation. Implied by RecordWildCards .
DuplicateRecordFields (page 278)	Allow definition of record types with identically-named fields.
EmptyCase (page 259)	Allow empty case alternatives.
EmptyDataDecls (page 264)	Allow definition of empty data types.
ExistentialQuantification (page 267)	Enable liberalised type synonyms.
ExplicitForAll (page 367)	Enable explicit universal quantification. Implied by ScopedTypeVariables .
ExplicitNamespaces (page 262)	Enable using the keyword type to specify the namespace of a type.
ExtendedDefaultRules (page 30)	Use GHCi’s extended default rules in a normal module.
FlexibleContexts (page 312)	Enable flexible contexts. Implied by ImplicitParams (page 377).
FlexibleInstances (page 321)	Enable flexible instances. Implies TypeSynonymInstances (page 324).
ForeignFunctionInterface (page 461)	Enable foreign function interface.
FunctionalDependencies (page 315)	Enable functional dependencies. Implies MultiParamTypeClasses (page 312).
GADTs (page 275)	Enable generalised algebraic data types. Implies GADTSyntax (page 271).
GADTSyntax (page 271)	Enable generalised algebraic data type syntax.
GeneralisedNewtypeDeriving (page 297)	Enable newtype deriving.
ImplicitParams (page 377)	Enable Implicit Parameters. Implies FlexibleContexts (page 312).
ImplicitPrelude (page 256)	Don’t implicitly import <code>Prelude</code> . Implied by RebindableSyntax (page 256).
ImpredicativeTypes (page 384)	Enable impredicative types. Implies RankNTypes (page 380).
IncoherentInstances (page 324)	Enable incoherent instances. Implies OverlappingInstances (page 324).
InstanceSigs (page 327)	Enable instance signatures.
InterruptibleFFI (page 462)	Enable interruptible FFI.
KindSignatures (page 369)	Enable kind signatures. Implied by TypeFamilies (page 334).
LambdaCase (page 259)	Enable lambda-case expressions.
LiberalTypeSynonyms (page 266)	Enable liberalised type synonyms.
MagicHash (page 241)	Allow <code>#</code> as a postfix modifier on identifiers.
MonadComprehensions (page 253)	Enable monad comprehensions.
MonadFailDesugaring (page 256)	Enable monadfail desugaring.
MonoLocalBinds (page 374)	Enable do not generalise local bindings. Implied by TypeFamilies (page 334).
MonomorphismRestriction (page 374)	Disable the monomorphism restriction.
MultiParamTypeClasses (page 312)	Enable multi parameter type classes. Implied by FunctionalDependencies (page 315).
MultiWayIf (page 260)	Enable multi-way if-expressions.
NamedFieldPuns (page 280)	Enable record puns.
NamedWildCards (page 389)	Enable named wildcards.
NegativeLiterals (page 242)	Enable support for negative literals.
NPlusKPatterns (page 245)	Enable support for n+k patterns. Implied by <code>Haskell98</code> .
NullaryTypeClasses (page 315)	Deprecated, does nothing. nullary (no parameter) type classes.
NumDecimals (page 242)	Enable support for ‘fractional’ integer literals.

Extension	Description
OverlappingInstances (page 324)	Enable overlapping instances.
OverloadedLabels (page 329)	Enable overloaded labels.
OverloadedLists (page 331)	Enable overloaded lists.
OverloadedStrings (page 328)	Enable overloaded string literals.
PackageImports (page 261)	Enable package-qualified imports.
ParallelListComp (page 251)	Enable parallel list comprehensions. Implied by ParallelArrows .
PartialTypeSignatures (page 387)	Enable partial type signatures.
PatternGuards (page 243)	Disable pattern guards. Implied by Haskell98 .
PatternSynonyms (page 305)	Enable pattern synonyms.
PolyKinds (page 352)	Enable kind polymorphism. Implies KindSignatures (page 352).
PostfixOperators (page 258)	Enable postfix operators.
QuasiQuotes (page 402)	Enable quasiquotation.
Rank2Types (page 380)	Enable rank-2 types. Synonym for RankNTypes (page 380).
RankNTypes (page 380)	Enable rank-N types. Implied by ImpredicativeTypes (page 380).
RebindableSyntax (page 256)	Employ rebinding syntax. Implies NoImplicitPrelude (page 256).
RecordWildCards (page 281)	Enable record wildcards. Implies DisambiguateRecordFields (page 281).
RecursiveDo (page 245)	Enable recursive do (mdo) notation.
RoleAnnotations (page 444)	Enable role annotations.
Safe (page 458)	Enable the Safe Haskell (page 449) Safe mode.
ScopedTypeVariables (page 370)	Enable lexically-scoped type variables.
StandaloneDeriving (page 287)	Enable standalone deriving.
StaticPointers (page 418)	Enable static pointers.
Strict (page 412)	Make bindings in the current module strict by default.
StrictData (page 412)	Enable default strict datatype fields.
TemplateHaskell (page 394)	Enable Template Haskell.
TemplateHaskellQuotes (page 395)	Enable quotation subset of Template Haskell (page 394).
TraditionalRecordSyntax (page 277)	Disable support for traditional record syntax (as supported by GHC 7.10).
TransformListComp (page 252)	Enable generalised list comprehensions.
Trustworthy (page 458)	Enable the Safe Haskell (page 449) Trustworthy mode.
TupleSections (page 258)	Enable tuple sections.
TypeApplications (page 375)	Enable type application syntax.
TypeFamilies (page 334)	Enable type families. Implies ExplicitNamespaces (page 262).
TypeFamilyDependencies (page 347)	Enable injective type families. Implies TypeFamilies (page 334).
TypeInType (page 352)	Allow kinds to be used as types, including explicit kind variables.
TypeOperators (page 265)	Enable type operators. Implies ExplicitNamespaces (page 262).
TypeSynonymInstances (page 320)	Enable type synonyms in instance heads. Implied by FlexibleInstances (page 320).
UnboxedSums (page 239)	Enable unboxed sums.
UnboxedTuples (page 238)	Enable the use of unboxed tuple syntax.
UndecidableInstances (page 322)	Enable undecidable instances.
UndecidableSuperClasses (page 333)	Allow all superclass constraints, including those that may require undecidable instances.
UnicodeSyntax (page 240)	Enable unicode syntax.
Unsafe (page 459)	Enable Safe Haskell (page 449) Unsafe mode.
ViewPatterns (page 243)	Enable view patterns.

Although not recommended, the deprecated [-fglasgow-exts](#) (page 235) 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 313)

- [DeriveDataTypeable](#) (page 295)
- [DeriveFoldable](#) (page 292)
- [DeriveFunctor](#) (page 289)
- [DeriveGeneric](#) (page 440)
- [DeriveTraversable](#) (page 294)
- [EmptyDataDecls](#) (page 264)
- [ExistentialQuantification](#) (page 267)
- [ExplicitNamespaces](#) (page 262)
- [FlexibleContexts](#) (page 312)
- [FlexibleInstances](#) (page 321)
- [ForeignFunctionInterface](#) (page 461)
- [FunctionalDependencies](#) (page 315)
- [GeneralizedNewtypeDeriving](#) (page 297)
- [ImplicitParams](#) (page 377)
- [KindSignatures](#) (page 369)
- [LiberalTypeSynonyms](#) (page 266)
- [MagicHash](#) (page 241)
- [MultiParamTypeClasses](#) (page 312)
- [ParallelListComp](#) (page 251)
- [PatternGuards](#) (page 243)
- [PostfixOperators](#) (page 258)
- [RankNTypes](#) (page 380)
- [RecursiveDo](#) (page 245)
- [ScopedTypeVariables](#) (page 370)
- [StandaloneDeriving](#) (page 287)
- [TypeOperators](#) (page 265)
- [TypeSynonymInstances](#) (page 320)
- [UnboxedTuples](#) (page 238)
- [UnicodeSyntax](#) (page 240)
- UnliftedFFITypes

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.

9.2 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 library `GHC.Prim`, for which there is detailed online documentation `<GHC.Prim.>`. (This documentation is generated from the file `compiler/prelude/primops.txt.pp`.)

If you want to mention any of the primitive data types or operations in your program, you must first import `GHC.Prim` to bring them into scope. Many of them have names ending in `#`, and to mention such names you need the [MagicHash](#) (page 241) extension.

The primops make extensive use of [unboxed types](#) (page 237) and [unboxed tuples](#) (page 238), which we briefly summarise here.

9.2.1 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 241)). For some primitive types we have special syntax for literals, also described in the [same section](#) (page 241).

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.

9.2.2 Unboxed type kinds

Because unboxed types are represented without the use of pointers, we cannot store them in use a polymorphic datatype at an unboxed 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 datatype). Accordingly, the *kind* of an unboxed type is different from the kind of a boxed type.

The Haskell Report describes that `*` is the kind of ordinary datatypes, such as `Int`. Furthermore, type constructors can have kinds with arrows; for example, `Maybe` has kind `* -> *`. 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 `*` is actually just a synonym for `TYPE 'PtrRepLifted`. More details of the `TYPE` mechanisms appear in the [section on runtime representation polymorphism](#) (page 361).

Given that `Int#`'s kind is not `*`, it then it follows that `Maybe Int#` is disallowed. Similarly, because type variables tend to be of kind `*` (for example, in `(.) :: (b -> c) -> (a -> b) -> a -> c`, all the type variables have kind `*`), 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#
```

- 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 *-Wunboxed-strict-patterns* (page 87).) 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#`.

9.2.3 Unboxed tuples

UnboxedTuples

Since 6.8.1

Unboxed tuples aren't really exported by `GHC.Exts`; they are a syntactic extension (*Unboxed-Tuples* (page 238)). 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 `I0` and `ST` monads use unboxed tuples to avoid unnecessary allocation during sequences of operations.

There are some restrictions on the use of unboxed tuples:

- 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.

9.2.4 Unboxed sums

UnboxedSums

Since 8.2.1

Enable the use of unboxed sum syntax.

`-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 tuple and sums).

Unboxed tuples can be used for multi-arity alternatives. For example:

```
(# (# Int, String #) | Bool #)
```

Term level syntax is similar. Leading and preceding bars (`|`) indicate which alternative it is. Here is two terms of the type shown above:

```
(# (# 1, "foo" #) | #) -- first alternative
(# | True #) -- second alternative
```

Pattern syntax reflects the term syntax:

```
case x of
  (# (# i, str #) | #) -> ...
  (# | bool #) -> ...
```

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. Code generator tries to generate as compact layout as possible. In the best case, size of an unboxed sum is size of its biggest alternative + one word (for tag). The algorithm for generating 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. E.g. say two alternatives have these fields:

`Word32, String, Float#
Float#, Float#, Maybe Int`

Final layout will be something like

`Int32, Float32, Float32, Word32, Pointer`

First *Int32* is for the tag. It has 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, and second alternative needs two *Float32* fields. *Word32* field is for the *Word32* in the first alternative. *Pointer* field is shared between *String* and *Maybe Int* values of the alternatives.

In the case of enumeration types (like *Bool*), the unboxed sum layout only has an *Int32* field (i.e. the whole thing is represented by an integer).

In the example above, a value of this type is thus represented as 5 values. As an another example, this is the layout for unboxed version of *Maybe a* type:

`Int32, Pointer`

The *Pointer* field is not used when tag says that it's *Nothing*. Otherwise *Pointer* points to the value in *Just*.

9.3 Syntactic extensions

9.3.1 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 240) 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

9.3.2 The magic hash

MagicHash

Since 6.8.1

Enables the use of the hash character (#) as an identifier suffix.

The language extension *MagicHash* (page 241) 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 241) extension bring anything into scope. For example, to bring `Int#` into scope you must import `GHC.Prim` (see *Unboxed types and primitive operations* (page 236)); the *MagicHash* (page 241) 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 241) also enables some new forms of literals (see *Unboxed types* (page 237)):

- `'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#`

9.3.3 Negative literals

NegativeLiterals

Since 7.8.1

Enable the use of un-parenthesized negative numeric literals.

The literal `-123` is, according to Haskell98 and Haskell 2010, desugared as `negate (fromInteger 123)`. The language extension *NegativeLiterals* (page 242) means that it is instead desugared as `fromInteger (-123)`.

This can make a difference 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.

9.3.4 Fractional looking integer literals

NumDecimals

Since 7.8.1

Allow the use of floating-point literal syntax for integral types.

Haskell 2010 and Haskell 98 define floating literals with the syntax `1.2e6`. These literals have the type `Fractional a => a`.

The language extension *NumDecimals* (page 242) allows you to also use the floating literal syntax for instances of `Integral`, and have values like `(1.2e6 :: Num a => a)`

9.3.5 Binary integer literals

BinaryLiterals

Since 7.10.1

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 242) 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 242) is enabled.

9.3.6 Hexadecimal floating point literals

-XHexFloatLiterals

Since 8.4.1

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 than 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`

9.3.7 Pattern guards

NoPatternGuards

Implied by `Haskell98`

Since 6.8.1

Disable [pattern guards](#).

9.3.8 View patterns

ViewPatterns

Since 6.10.1

Allow use of view pattern syntax.

View patterns are enabled by the language extension [ViewPatterns](#) (page 243). 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 243) 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 [Trac #4061](#).)

- **Typing:** If $\langle \text{exp} \rangle$ has type $\langle T1 \rangle \rightarrow \langle T2 \rangle$ and $\langle \text{pat} \rangle$ matches a $\langle T2 \rangle$, then the whole view pattern matches a $\langle T1 \rangle$.
- **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 $\langle v \rangle$ against a pattern $(\langle \text{exp} \rangle \rightarrow \langle \text{pat} \rangle)$, evaluate $(\langle \text{exp} \rangle \langle v \rangle)$ and match the result against $\langle \text{pat} \rangle$.

- **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 $(x, \text{view } x \rightarrow y)$ will not be coalesced.

9.3.9 n+k patterns

NPlusKPatterns

Implied by Haskell98

Since 6.12.1

Enable use of $n+k$ patterns.

9.3.10 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 245) (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.

Recursive binding groups

The extension [RecursiveDo](#) (page 245) 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
    ; let { r1 = f a r2
          ;   ; r2 = g r1 }
    ; return (r1 ++ r2) }      do { a <- getChar
                                ; rec { r1 <- f a r2
                                      ;   ; r2 <- g r1 }
                                ; 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
    ; c <- f b a }      ==>    (b,c) <- mfix (\ ~(b,c) -> do { b <- f a c
                                                            ; 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.

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 245), 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.)

9.3.11 Applicative do-notation

ApplicativeDo

Since 8.0.1

Allow use of Applicative do notation.

The language option [ApplicativeDo](#) (page 248) 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 248) without fear of breaking your program. There is one pitfall to watch out for; see [Things to watch out for](#) (page 250).

There are no syntactic changes with [ApplicativeDo](#) (page 248). 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 $(\backslash x \rightarrow \text{not } x) <\$> m$. 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 198), 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 $p1 \dots pn$ are mentioned in $E1 \dots En$, and $p1 \dots pn$ are all variables or lazy patterns, then the expression will only require Applicative. Otherwise, the expression will require Monad. The block may return a pure expression E depending upon the results $p1 \dots pn$ 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 $O(n^2)$.

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.

Things to watch out for

Your code should just work as before when [ApplicativeDo](#) (page 248) 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
```

9.3.12 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.

9.3.13 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 252).

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 `then` 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", "hellohell", "hellohell", "hellohell", "hellohell"]
```

9.3.14 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 251)) and transform comprehensions (*Generalised (SQL-like) List Comprehensions* (page 252)) 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 a `MonadPlus` instance:

```
do x <- [1..10]
  guard (x <= 5)
  return x
```

- Transform statements (as with *TransformListComp* (page 252)):

```
[ 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 252)):

```
[ 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 251)):

```
[ (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 253) 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 252) and *Parallel List Comprehensions* (page 251). 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 253) backward compatible to built-in, transform and parallel list comprehensions.

More formally, the desugaring is as follows. We write $D[e \mid Q]$ 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 ]

-- 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 252)) 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 256)). 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).

9.3.15 New monadic failure desugaring mechanism

MonadFailDesugaring

Since 8.0.1

Use the `MonadFail.fail` instead of the legacy `Monad.fail` function when desugaring refutable patterns in `do` blocks.

The `-XMonadFailDesugaring` extension switches the desugaring of `do`-blocks to use `MonadFail.fail` instead of `Monad.fail`. This will eventually be the default behaviour in a future GHC release, under the [MonadFail Proposal \(MFP\)](#).

This extension is temporary, and will be deprecated in a future release. It is included so that library authors have a hard check for whether their code will work with future GHC versions.

9.3.16 Rebindable syntax and the implicit Prelude import

NoImplicitPrelude

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. (But don't call it `Prelude`; the Haskell module namespace is flat, and you must not conflict with any `Prelude` module.)

RebindableSyntax

Implies [NoImplicitPrelude](#) (page 256)

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 256) 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 handed in just the same way, except that the translation is from-`Rational (3.68::Rational)`.
- 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 245)), and parallel array comprehensions, are unaffected.
- Arrow notation (see *Arrow notation* (page 404)) 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 331).
- An overloaded label `“#foo”` means `“fromLabel @"foo”`, rather than `“GHC.OverloadedLabels.fromLabel @"foo”` (see *Overloaded labels* (page 329)).

RebindableSyntax (page 256) implies *NoImplicitPrelude* (page 256).

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.

Things unaffected by RebindableSyntax

RebindableSyntax (page 256) 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 256) and *OverloadedStrings* (page 328) 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 256) behavior within derived code, so GHC simply ignores *RebindableSyntax* (page 256) entirely when checking derived code.

9.3.17 Postfix operators

PostfixOperators

Since 7.10.1

Allow the use of post-fix operators

The *PostfixOperators* (page 258) extension enables a small extension to the syntax of left operator sections, which allows you to define postfix operators. The extension is this: the left section

```
(e !)
```

is equivalent (from the point of view of both type checking and execution) to the expression

```
((!) e)
```

(for any expression *e* and operator *(!)*). 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.

9.3.18 Tuple sections

TupleSections

Since 6.12

Allow the use of tuple section syntax

The *TupleSections* (page 258) 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 238) 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.

9.3.19 Lambda-case

LambdaCase

Since 7.6.1

Allow the use of lambda-case syntax.

The *LambdaCase* (page 259) extension enables expressions of the form

```
\case { p1 -> e1; ...; pN -> eN }
```

which is equivalent to

```
\freshName -> case freshName of { p1 -> e1; ...; pN -> eN }
```

Note that `\case` starts a layout, so you can write

```
\case
  p1 -> e1
  ...
  pN -> eN
```

9.3.20 Empty case alternatives

EmptyCase

Since 7.8.1

Allow empty case expressions.

The *EmptyCase* (page 259) extension enables case expressions, or lambda-case expressions, that have no alternatives, thus:

```
case e of { }    -- No alternatives
```

or

```
\case { }      -- -XLambdaCase is also required
```

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 [Trac #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 81) and *-Werror* (page 75). On the other hand (A) looks dangerous, and GHC doesn't check to make sure that, in fact, the function can never get called.

9.3.21 Multi-way if-expressions

MultiWayIf

Since 7.6.1

Allow the use of multi-way-if syntax.

With *MultiWayIf* (page 260) 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.

9.3.22 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 `let`. Or, in recursive `do` expressions (*The recursive `do`-notation* (page 245)), 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.

9.3.23 Import and export extensions

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 79), which is off by default but included with *-W* (page 75), warns if you hide something that the imported module does not export.

Package-qualified imports

PackageImports

Since 6.10.1

Allow the use of package-qualified import syntax.

With the [PackageImports](#) (page 261) 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 167) for an alternative way of disambiguating between module names.

Safe imports

Safe

Since 7.2.1

Declare the Safe Haskell state of the current module.

Trustworthy

Since 7.2.1

Declare the Safe Haskell state of the current module.

Unsafe

Since 7.4.1

Declare the Safe Haskell state of the current module.

With the [Safe](#) (page 458), [Trustworthy](#) (page 458) and [Unsafe](#) (page 459) 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 449).

Explicit namespaces in import/export

ExplicitNamespaces

Since 7.6.1

Enable use of explicit namespaces in module export lists.

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 265)) it becomes possible to declare `(++)` as a *type constructor*. In that case, how would you export or import it?

The [ExplicitNamespaces](#) (page 262) 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 262) is implied by [TypeOperators](#) (page 265) and (for some reason) by [TypeFamilies](#) (page 334).

In addition, with [PatternSynonyms](#) (page 305) 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 309)).

9.3.24 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: [ExplicitForAll](#) (page 367), and hence by [ScopedTypeVariables](#) (page 370), [LiberalTypeSynonyms](#) (page 266), [RankNTypes](#) (page 380), [ExistentialQuantification](#) (page 267)

mdo Stolen by: [RecursiveDo](#) (page 245)

foreign Stolen by: [ForeignFunctionInterface](#) (page 461)

rec, proc, -<, >-, -<<, >>-, (|, |) Stolen by: [Arrows](#) (page 404)

?varid Stolen by: [ImplicitParams](#) (page 377)

[|, [e|, [p|, [d|, [t|, [|, [e|] Stolen by: [QuasiQuotes](#) (page 402). Moreover, this introduces an ambiguity with list comprehension syntax. See the [discussion on quasi-quoting](#) (page 403) for details.

\$(, \$\$, \$varid, \$\$varid Stolen by: [TemplateHaskell](#) (page 394)

[varid| Stolen by: [QuasiQuotes](#) (page 402)

`<varid>`, `#(char)`, `#`, `<string>`, `#`, `<integer>`, `#`, `<float>`, `#`, `<float>`, `##` Stolen by: [MagicHash](#) (page 241)

`(#, #)` Stolen by: [UnboxedTuples](#) (page 238)

`<varid>`, `!`, `<varid>` Stolen by: [BangPatterns](#) (page 411)

`pattern` Stolen by: [PatternSynonyms](#) (page 305)

9.4 Extensions to data types and type synonyms

9.4.1 Data types with no constructors

`EmptyDataDecls`

Since 6.8.1

Allow definition of empty data types.

With the [EmptyDataDecls](#) (page 264) extension, GHC lets you declare a data type with no constructors. For example:

```
data S      -- S :: *
data T a    -- T :: * -> *
```

Syntactically, the declaration lacks the “= constrs” part. The type can be parameterised over types of any kind, but if the kind is not `*` then an explicit kind annotation must be used (see [Explicitly-kinded quantification](#) (page 369)).

Such data types have only one value, namely bottom. Nevertheless, they can be useful when defining “phantom types”.

In conjunction with the [-XEmptyDataDeriving](#) (page 286) extension, empty data declarations can also derive instances of standard type classes (see [Deriving instances for empty data types](#) (page 286)).

9.4.2 Data type contexts

`DatatypeContexts`

Since 7.0.1

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`.

9.4.3 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 `:+:`. `Int `a` Bool`.

- Function arrow is `infixr` with fixity 0 (this might change; it's not clear what it should be).

9.4.4 Type operators

TypeOperators

Implies [ExplicitNamespaces](#) (page 262)

Since 6.8.1

Allow the use and definition of types with operator names.

In types, an operator symbol like `(+)` is normally treated as a type *variable*, just like `a`. Thus in Haskell 98 you can say

```
type T (+) = ((+), (+))
-- Just like: type T a = (a,a)

f :: T Int -> Int
f (x,y) = x
```

As you can see, using operators in this way is not very useful, and Haskell 98 does not even allow you to write them infix.

The language *TypeOperators* (page 265) changes this behaviour:

- Operator symbols become type *constructors* rather than type *variables*.
- 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
```

- 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 262) (which is implied by *TypeOperators* (page 265)) 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 262).

- 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 265), the function and type constructor share a single fixity.

9.4.5 Liberalised type synonyms

LiberalTypeSynonyms

Implies *ExplicitForAll* (page 367)

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 266) 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 write a `forall` (including overloading) in a type synonym, thus:

```
type Discard a = forall b. Show b => a -> b -> (a, String)

f :: Discard a
f x y = (x, show y)

g :: Discard Int -> (Int,String)    -- A rank-2 type
g f = f 3 True
```

- If you also use *UnboxedTuples* (page 238), you can write an unboxed tuple in a type synonym:

```
type Pr = (# Int, Int #)

h :: Int -> Pr
h x = (# x, x #)
```

- 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 currently does kind checking before expanding synonyms (though even that could be changed).

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-all (if *ImpredicativeTypes* (page 384) 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.

9.4.6 Existentially quantified data constructors

ExistentialQuantification

Implies *ExplicitForAll* (page 367)

Since 6.8.1

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 Odierky (*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.

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.

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.

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`. 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   (f2's type mentions c, which is not a simple
                        --        type-variable argument in G1's result type)

```

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!

9.4.7 Declaring data types with explicit constructor signatures

GADTSyntax

Since 7.2.1

Allow the use of GADT syntax in data type definitions (but not GADTs themselves; for this see [GADTs](#) (page 275))

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
```

The form is called a “GADT-style declaration” because Generalised Algebraic Data Types, described in [Generalised Algebraic Data Types \(GADTs\)](#) (page 275), can only be declared using this form.

Notice that GADT-style syntax generalises existential types ([Existentially quantified data constructors](#) (page 267)). For example, these two declarations are equivalent:

```
data Foo = forall a. MkFoo a (a -> Bool)
data Foo' where { MkFoo' :: a -> (a->Bool) -> Foo' }
```

Any data type 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 268). 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.

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 275)).
- 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 :: * -> * where ...
```

or even a mixture of the two:

```
data Bar a :: (* -> *) -> * 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 :: * -> *) 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 )

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 267).

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 269)), 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
```

9.4.8 Generalised Algebraic Data Types (GADTs)

GADTs

Implies *MonoLocalBinds* (page 374), *GADTSyntax* (page 271)

Since 6.8.1

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 275). The [GADTs](#) (page 275) extension also sets [GADTSyntax](#) (page 271) and [MonoLocalBinds](#) (page 374).

- A GADT can only be declared using GADT-style syntax ([Declaring data types with explicit constructor signatures](#) (page 271)); 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).
- 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.
- You cannot use a deriving clause for a GADT; only for an ordinary data type.
- As mentioned in [Declaring data types with explicit constructor signatures](#) (page 271), 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
```

- 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 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.

9.5 Extensions to the record system

9.5.1 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 277) extension.

9.5.2 Record field disambiguation

DisambiguateRecordFields

Since 6.8.1

Since 6.8.1

Allow the compiler to automatically choose between identically-named record selectors based on type (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

  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 277) 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.

Some details:

- Field disambiguation can be combined with punning (see [Record puns](#) (page 280)). For example:

```
module Foo where
  import M
  x=True
  ok3 (MkS { x }) = x+1    -- Uses both disambiguation and punning
```

- With [DisambiguateRecordFields](#) (page 277) 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).

9.5.3 Duplicate record fields

DuplicateRecordFields

Implies [DisambiguateRecordFields](#) (page 277)

Since 8.0.1

Allow definition of record types with identically-named fields.

Going beyond [DisambiguateRecordFields](#) (page 277) (see [Record field disambiguation](#) (page 277)), the [DuplicateRecordFields](#) (page 278) 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 277)):

```
s = MkS { x = 3 }
f (MkT { x = b }) = b
```

Field names used as selector functions or in record updates must be unambiguous, either because there is only one such field in scope, or because a type signature is supplied, as described in the following sections.

Selector functions

Fields may be used as selector functions only if they are unambiguous, so this is still not allowed if both S(x) and T(x) are in scope:

```
bad r = x r
```

An ambiguous selector may be disambiguated by the type being “pushed down” to the occurrence of the selector (see [Type inference](#) (page 382) for more details on what “pushed down” means). For example, the following are permitted:

```
ok1 = x :: S -> Int

ok2 :: S -> Int
ok2 = x

ok3 = k x -- assuming we already have k :: (S -> Int) -> _
```

In addition, the datatype that is meant may be given as a type signature on the argument to the selector:

```
ok4 s = x (s :: S)
```

However, we do not infer the type of the argument to determine the datatype, or have any way of deferring the choice to the constraint solver. Thus the following is ambiguous:

```
bad :: S -> Int
bad s = x s
```

Even though a field label is duplicated in its defining module, it may be possible to use the selector unambiguously elsewhere. For example, another module could import `S(x)` but not `T(x)`, and then use `x` unambiguously.

Record updates

In a record update such as `e { x = 1 }`, if there are multiple `x` fields in scope, then the type of the context must fix which record datatype is intended, or a type annotation must be supplied. Consider the following definitions:

```
data S = MkS { foo :: Int }
data T = MkT { foo :: Int, bar :: Int }
data U = MkU { bar :: Int, baz :: Int }
```

Without [DuplicateRecordFields](#) (page 278), an update mentioning `foo` will always be ambiguous if all these definitions were in scope. When the extension is enabled, there are several options for disambiguating updates:

- Check for types that have all the fields being updated. For example:

```
f x = x { foo = 3, bar = 2 }
```

Here `f` must be updating `T` because neither `S` nor `U` have both fields.

- Use the type being pushed in to the record update, as in the following:

```
g1 :: T -> T
g1 x = x { foo = 3 }

g2 x = x { foo = 3 } :: T

g3 = k (x { foo = 3 }) -- assuming we already have k :: T -> _
```

- Use an explicit type signature on the record expression, as in:

```
h x = (x :: T) { foo = 3 }
```

The type of the expression being updated will not be inferred, and no constraint-solving will be performed, so the following will be rejected as ambiguous:

```
let x :: T
    x = blah
in x { foo = 3 }

\x -> [x { foo = 3 }, blah :: T ]

\ (x :: T) -> x { foo = 3 }
```

Import and export of record fields

When *DuplicateRecordFields* (page 278) 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 ...
```

Similar restrictions apply on import.

9.5.4 Record puns

NamedFieldPuns

Since 6.10.1

Allow use of record puns.

Record puns are enabled by the language extension *NamedFieldPuns* (page 280).

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.)

9.5.5 Record wildcards

RecordWildCards

Implies [DisambiguateRecordFields](#) (page 277).

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 281). This extension implies [DisambiguateRecordFields](#) (page 277).

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 280)); 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(a,c) )
    f b = R { .. }
```

The `R{..}` expands to `R{M.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 use 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)
```


9.5.6 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 283)). Users may define their own instances of `HasField` also (see [Virtual record fields](#) (page 284)).

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 RankTypes #-}
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
```

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.

9.6 Extensions to the “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 287).
- In Haskell 98, the only derivable classes are `Eq`, `Ord`, `Enum`, `Ix`, `Bounded`, `Read`, and `Show`. *Various language extensions* (page 288) 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 297) for newtypes and
 - *deriving any class* (page 302) using an empty instance declaration.

The user can optionally declare the desired *deriving strategy* (page ??), especially if the compiler chooses the wrong one *by default* (page 305).

9.6.1 Deriving instances for empty data types

-XEmptyDataDeriving

Since 8.4.1

Allow deriving instances of standard type classes for empty data types.

One can write data types with no constructors using the [-XEmptyDataDecls](#) (page 264) flag (see [Data types with no constructors](#) (page 264)), 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 [-XEmptyDataDeriving](#) (page 286) 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 [-XEmptyDataDeriving](#) (page 286) 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 [-XEmptyDataDeriving](#) (page 286) to be used in conjunction with empty data types. These include:

- [-XStandaloneDeriving](#) (page 287) (see [Stand-alone deriving declarations](#) (page 287))
- Type classes which require their own extensions to be enabled to be derived, such as [-XDeriveFunctor](#) (page 289) (see [Deriving instances of extra classes \(Data, etc.\)](#) (page 288))
- [-XDeriveAnyClass](#) (page 302) (see [Deriving any other class](#) (page 302))

9.6.2 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 287).

9.6.3 Stand-alone deriving declarations

StandaloneDeriving

Since 6.8.1

Allow the use of stand-alone deriving declarations.

GHC allows stand-alone deriving declarations, enabled by *StandaloneDeriving* (page 287):

```
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 162)).
- 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.)
- 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 321), *Relaxed rules for instance contexts* (page 321)). 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 302), since deriving an instance via [DeriveAnyClass](#) (page 302) simply generates an empty instance declaration, which does not require the use of any constructors. See the [deriving any class](#) (page 302) 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](#) (page 320).
- 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 297)). 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.

9.6.4 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 440), 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 439).
- With [DeriveFunctor](#) (page 289), you can derive instances of the class `Functor`, defined in `GHC.Base`.
- With [DeriveDataTypeable](#) (page 295), you can derive instances of the class `Data`, defined in `Data.Data`.
- With [DeriveFoldable](#) (page 292), you can derive instances of the class `Foldable`, defined in `Data.Foldable`.

- With [DeriveTraversable](#) (page 294), 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 294) implies [DeriveFunctor](#) (page 289) and [DeriveFoldable](#) (page 292).
- With [DeriveLift](#) (page 296), 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 287)).

In each case the appropriate class must be in scope before it can be mentioned in the deriving clause.

Deriving Functor instances

DeriveFunctor

Since 7.10.1

Allow automatic deriving of instances for the `Functor` typeclass.

With [DeriveFunctor](#) (page 289), one can derive `Functor` instances for data types of kind `* -> *`. 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 289) 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 289) 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 = 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 289) 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 289) 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 `CovFuns` 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 264) constraint (e.g., `data Ord a => 0 a = 0 a`).
3. A data type's last type variable is used in an *ExistentialQuantification* (page 267) 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 442)), 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 259).

```
data V a deriving Functor
type role V nominal
```

will produce

```
instance Functor V where fmap _ z = case z of
```

Deriving Foldable instances

DeriveFoldable

Since 7.10.1

Allow automatic deriving of instances for the Foldable typeclass.

With [DeriveFoldable](#) (page 292), one can derive Foldable instances for data types of kind `*` `-> *`. 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 292) is adapted from the [DeriveFunctor](#) (page 289) algorithm, but it generates definitions for `foldMap`, `foldr`, and `null` instead of `fmap`. In addition, [DeriveFoldable](#) (page 292) 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 442)), [DeriveFoldable](#) (page 292) 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 292) 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 289) would generate `f a` for an `fmap` definition. [DeriveFoldable](#) (page 292) 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 289) recursively calls `fmap` on it. Similarly, *DeriveFoldable* (page 292) 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 289) puts everything back together again at the end by invoking the constructor. *DeriveFoldable* (page 292), 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
```

```
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 $(a \sim \text{Int})$, `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`.

Deriving Traversable instances

DeriveTraversable

Implies [DeriveFoldable](#) (page 292), [DeriveFunctor](#) (page 289)

Since 7.10.1

Allow automatic deriving of instances for the `Traversable` typeclass.

With [DeriveTraversable](#) (page 294), one can derive `Traversable` instances for data types of kind `* -> *`. 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 294) is adapted from the [DeriveFunctor](#) (page 289) algorithm, but it generates a definition for `traverse` instead of `fmap`. In addition, [DeriveTraversable](#) (page 294) 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 442)), [DeriveTraversable](#) (page 294) 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 294) will derive the laziest instance it can.

```
data V a deriving Traversable
type role V nominal
```

will generate the following, using *EmptyCase* (page 259):

```
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 289) and *DeriveTraversable* (page 294) 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 289) recursively calls `fmap` on it. Similarly, *DeriveTraversable* (page 294) would recursively call `traverse`.
3. *DeriveFunctor* (page 289) puts everything back together again at the end by invoking the constructor. *DeriveTraversable* (page 294) does something similar, but it works in an Applicative context by chaining everything together with `(<*>)`.

Unlike *DeriveFunctor* (page 289), *DeriveTraversable* (page 294) 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 289), *DeriveFoldable* (page 292), and *DeriveTraversable* (page 294), see [this wiki page](#).

Deriving Data Instances

DeriveDataTypeable

Since 6.8.1

Enable automatic deriving of instances for the `Data` typeclass

Deriving Typeable instances

The class `Typeable` is very special:

- `Typeable` is kind-polymorphic (see *Kind polymorphism and Type-in-Type* (page 352)).
- 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 295) 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
```

Deriving Lift instances

DeriveLift

Since 7.2.1

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`), 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)
    = appE
      (conE
        (mkNameG_d "package-name" "Bar" "Foo"))
      (lift a)
  lift (u ^:: v)
    = infixApp
      (lift u)
      (conE
        (mkNameG_d "package-name" "Bar" " ^:: "))
      (lift v)
-}

-----
{-# LANGUAGE TemplateHaskell #-}
module Baz where

import Bar
import Language.Haskell.TH.Lift

foo :: Foo String
foo = $(lift $ Foo "foo")

fooExp :: Lift a => Foo a -> Q Exp
fooExp f = [| f |]
```

DeriveLift (page 296) also works for certain unboxed types (`Addr#`, `Char#`, `Double#`, `Float#`, `Int#`, and `Word#`):

```
{-# 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)
    = appE
      (conE
       (mkNameG_d "package-name" "Unboxed" "IntHash"))
      (litE
       (intPrimL (toInteger (I# i))))
-}
```

9.6.5 Generalised derived instances for newtypes

GeneralisedNewtypeDeriving
GeneralizedNewtypeDeriving

Since 6.8.1

Enable GHC's cunning generalised deriving mechanism for newtypes

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!

Generalising the deriving clause

GHC now permits such instances to be derived instead, using the extension *Generalized-NewtypeDeriving* (page 297), 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 299) 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 new type, 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.

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 305).
- 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 442).)
- C is allowed to have associated type families, provided they meet the requirements laid out in the section on [GND and associated types](#) (page 300).

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.)

Associated type families

GeneralizedNewtypeDeriving (page 297) 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 297) 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 322) 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 322) even if the generated code won't put the typechecker into a loop. For example:

```
instance C Int where
  type C Int = Int

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 322) 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!

9.6.6 Deriving any other class

DeriveAnyClass

Since 7.10.1

Allow use of any typeclass in deriving clauses.

With [DeriveAnyClass](#) (page 302) 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 422) is empty, and especially when writing *generic functions* (page 439).

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 302) 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 314) 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 297) is also on, [DeriveAnyClass](#) (page 302) 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 302):

```
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 302) 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 302) 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.

9.6.7 Deriving strategies

DerivingStrategies

Since 8.2.1

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 297) and [DeriveAnyClass](#) (page 302) 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 304) 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 304) 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 302)
- `newtype`: Use *GeneralizedNewtypeDeriving* (page 297)

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 288), 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 other any type class:
 - (a) When `DeriveAnyClass` is enabled, use `anyclass`.
 - (b) When `GeneralizedNewtypeDeriving` 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.

9.7 Pattern synonyms

PatternSynonyms

Since 7.8.1

Allow the definition of pattern synonyms.

Pattern synonyms are enabled by the language extension *PatternSynonyms* (page 305), which is required for defining them, but *not* for using them. More information and examples of view patterns can be found on the *Wiki page* [<PatternSynonyms>](#).

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 _                   = []

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 411) 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
```

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)

9.7.1 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	<code>zero = Point 0 0</code>
As a constructor with record syntax	<code>zero = Point { x = 0, y = 0 }</code>
In a pattern context	<code>isZero (Point 0 0) = True</code>
In a pattern context with record syntax	<code>isZero (Point { x = 0, y = 0 })</code>
In a pattern context with field puns	<code>getX (Point {x}) = x</code>
In a record update	<code>(0, 0) { x = 1 } == (1,0)</code>
Using record selectors	<code>x (0,0) == 0</code>

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.

9.7.2 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
```

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	<code>Name args</code>
Infix	<code>arg1 `Name` arg2</code> or <code>arg1 op arg2</code>
Record	<code>Name{arg1,arg2,...,argn}</code>

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 430) can be specified in order to tell the pattern match exhaustiveness checker that a set of pattern synonyms is complete.

9.7.3 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.

To bundle a pattern synonym with a type constructor, 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](#).

9.7.4 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 $\langle \text{CReq} \rangle$ and $\langle \text{CProv} \rangle$ are type contexts, and $\langle t1 \rangle$, $\langle t2 \rangle$, ..., $\langle tN \rangle$ and $\langle t \rangle$ are types. Notice the unusual form of the type, with two contexts $\langle \text{CReq} \rangle$ and $\langle \text{CProv} \rangle$:

- $\langle \text{CReq} \rangle$ are the constraints *required* to match the pattern.
- $\langle \text{CProv} \rangle$ 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 $(\text{Num } a, \text{Eq } a)$, so they appear in `f1`'s type. The call to `show` generates a $(\text{Show } b)$ constraint, where `b` is an existentially type variable bound by the pattern match on `MkT`. But the same pattern match also *provides* the constraint $(\text{Show } b)$ (see `MkT`'s type), and so all is well.

Exactly the same reasoning applies to `ExNumPat`: matching against `ExNumPat` *requires* the constraints $(\text{Num } a, \text{Eq } a)$, and *provides* the constraint $(\text{Show } b)$.

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
```

- 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 GHCi `:info` (page 47) command shows pattern types in this format.
- 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

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 [Trac #9953](#) for discussion of this choice.)

9.7.5 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. 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
```

9.8 Class and instances declarations

9.8.1 Class declarations

This section, and the next one, 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).

Multi-parameter type classes

MultiParamTypeClasses

Implies [ConstrainedClassMethods](#) (page 313)

Since 6.8.1

Allow the definition of typeclasses with more than one parameter.

Multi-parameter type classes are permitted, with extension [MultiParamTypeClasses](#) (page 312). For example:

```
class Collection c a where
  union :: c a -> c a -> c a
  ...etc.
```

The superclasses of a class declaration

FlexibleContexts

Since 6.8.1

Allow the use of complex constraints in class declaration contexts.

In Haskell 98 the context of a class declaration (which introduces superclasses) must be simple; that is, each predicate must consist of a class applied to type variables. The extension [FlexibleContexts](#) (page 312) ([The context of a type signature](#) (page 367)) lifts this restriction, so that the only restriction on the context in a class declaration is that the class hierarchy must be acyclic. So these class declarations are OK:

```
class Functor (m k) => FiniteMap m k where
  ...
```

```
class (Monad m, Monad (t m)) => Transform t m where
  lift :: m a -> (t m) a
```

As in Haskell 98, the class hierarchy must be acyclic. However, the definition of “acyclic” involves only the superclass relationships. For example, this is okay:

```
class C a where
  op :: D b => a -> b -> b

class C a => D a where ...
```

Here, C is a superclass of D, but it's OK for a class operation op of C to mention D. (It would not be OK for D to be a superclass of C.)

With the extension that adds a *kind of constraints* (page 366), you can write more exotic superclass definitions. The superclass cycle check is even more liberal in these case. For example, this is OK:

```
class A cls c where
  meth :: cls c => c -> c

class A B c => B c where
```

A superclass context for a class C is allowed if, after expanding type synonyms to their right-hand-sides, and uses of classes (other than C) to their superclasses, C does not occur syntactically in the context.

Constrained class method types

ConstrainedClassMethods

Since 6.8.1

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:

```
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:

```
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:

```
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 313). The restriction is a pretty stupid one in the first place, so *ConstrainedClassMethods* (page 313) is implied by *MultiParamTypeClasses* (page 312).

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 lift this restriction and give another type to the default method using the extension *DefaultSignatures* (page 314). For instance, if you have written a generic implementation of enumeration in a class `GEnum` with method `genum` in terms of `GHC.Generics`, you can specify a default method that uses that generic implementation:

```
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 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
```

`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 314), GHC implicitly fills in the default implementation like this:

```
instance Foo Int where
  bar = default_bar @Int
```

Where `@Int` utilizes visible type application (*Visible type application* (page 375)) to instantiate the `b` in `default bar :: forall b. C' => a -> b -> b`. In order for this type application to work, the default type signature for `bar` must have the same type variable order as the non-default signature! But there is no obligation for `C` and `C'` to be the same (see, for instance, the `Enum` example above, which relies on this).

To further explain this example, the right-hand side of the default type signature for `bar` must be something that is alpha-equivalent to `forall b. a -> b -> b` (where `a` is bound by the class itself, and is thus free in the methods' type signatures). So this would also be an acceptable default type signature:

```
default bar :: forall x. C' => a -> x -> x
```

But not this (since the free variable `a` is in the wrong place):

```
default bar :: forall b. C' => b -> a -> b
```

Nor this, since we can't match the type variable `b` with the concrete type `Int`:

```
default bar :: C' => a -> Int -> Int
```

That last one deserves a special mention, however, since `a -> Int -> Int` is a straightforward instantiation of `forall b. a -> b -> b`. You can still write such a default type signature, but you now must use type equalities to do so:

```
default bar :: forall b. (C', b ~ Int) => a -> b -> b
```

We use default signatures to simplify generic programming in GHC (*Generic programming* (page 439)).

Nullary type classes

NullaryTypeClasses

Since 7.8.1

Allows the use definition of type classes with no parameters. This extension has been replaced by *MultiParamTypeClasses* (page 312).

Nullary (no parameter) type classes are enabled with *MultiParamTypeClasses* (page 312); historically, they were enabled with the (now deprecated) *NullaryTypeClasses* (page 315). 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
```

9.8.2 Functional dependencies

FunctionalDependencies

Implies *MultiParamTypeClasses* (page 312)

Since 6.8.1

Allow use of functional dependencies in class declarations.

Functional dependencies are implemented as described by Mark Jones in *[Jones2000]* (page 521).

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](#).

Rules for functional dependencies

In a class declaration, all of the class type variables must be reachable (in the sense mentioned in *The context of a type signature* (page 367)) 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
```

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 ...
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 Odgersky, 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 [\[Jones1999\]](#) (page 521), 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`, ..., `ym` 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 `*`; 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.

9.8.3 Instance declarations

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.

Instance resolution

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 324) for flags that loosen the instance resolution rules.

Relaxed rules for the instance head

TypeSynonymInstances

Since 6.8.1

Allow definition of type class instances for type synonyms.

FlexibleInstances

Implies [TypeSynonymInstances](#) (page 320)

Since 6.8.1

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 320) 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 321) 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 324).

The [FlexibleInstances](#) (page 321) extension implies [TypeSynonymInstances](#) (page 320).

However, the instance declaration must still conform to the rules for instance termination: see [Instance termination rules](#) (page 322).

Relaxed rules for instance contexts

In Haskell 98, the class constraints in the context of the instance declaration must be of the form `C a` where `a` is a type variable that occurs in the head.

The [FlexibleContexts](#) (page 312) extension relaxes this rule, as well as relaxing the corresponding rule for type signatures (see [The context of a type signature](#) (page 367)). Specifically, [FlexibleContexts](#) (page 312), allows (well-kinded) class constraints of form `(C t1 ... tn)` in the context of an instance declaration.

Notice that the extension does not affect equality constraints in an instance context; they are permitted by [TypeFamilies](#) (page 334) or [GADTs](#) (page 275).

However, the instance declaration must still conform to the rules for instance termination: see [Instance termination rules](#) (page 322).

Instance termination rules

UndecidableInstances

Since 6.8.1

Permit definition of instances which may lead to type-checker non-termination.

Regardless of *FlexibleInstances* (page 321) and *FlexibleContexts* (page 312), instance declarations must conform to some rules that ensure that instance resolution will terminate. The restrictions can be lifted with *UndecidableInstances* (page 322) (see *Undecidable instances* (page 323)).

The rules are these:

1. The Paterson Conditions: for each class constraint $(C \text{ } t_1 \dots t_n)$ in the context
 - (a) No type variable has more occurrences in the constraint than in the head
 - (b) The constraint has fewer constructors and variables (taken together and counting repetitions) than the head
 - (c) 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
2. The Coverage Condition. For each functional dependency, $\langle \text{tvs} \rangle_{\text{left}} \rightarrow \langle \text{tvs} \rangle_{\text{right}}$, of the class, every type variable in $S(\langle \text{tvs} \rangle_{\text{right}})$ must appear in $S(\langle \text{tvs} \rangle_{\text{left}})$, 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.

A useful idiom permitted by the above rules is as follows. If one allows overlapping instance declarations then it's quite convenient to have a “default instance” declaration that applies if something more specific does not:

```
instance C a where
  op = ... -- Default
```

Undecidable instances

Sometimes even the termination rules of *Instance termination rules* (page 322) are too onerous. So GHC allows you to experiment with more liberal rules: if you use the experimental extension *UndecidableInstances* (page 322), both the Paterson Conditions and the Coverage Condition (described in *Instance termination rules* (page 322)) are lifted. Termination is still ensured by having a fixed-depth recursion stack. If you exceed the stack depth you get a sort of backtrace, and the opportunity to increase the stack depth with `-freduction-depth=(n)`. However, if you should exceed the default reduction depth limit, it is probably best just to disable depth checking, with `-freduction-depth=0`. The exact depth your program requires depends on minutiae of your code, and it may change between minor GHC releases. The safest bet for released code – if you're sure that it should compile in finite time – is just to disable the check.

For example, sometimes you might want to use the following to get the effect of a “class synonym”:

```
class (C1 a, C2 a, C3 a) => C a where { }

instance (C1 a, C2 a, C3 a) => C a where { }
```

This allows you to write shorter signatures:

```
f :: C a => ...
```

instead of

```
f :: (C1 a, C2 a, C3 a) => ...
```

The restrictions on functional dependencies (*Functional dependencies* (page 315)) 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 (some-what 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)`.

The *UndecidableInstances* (page 322) extension is also used to lift some of the restrictions imposed on type family instances. See *Decidability of type synonym instances* (page 340).

Overlapping instances

OverlappingInstances

Deprecated extension to weaken checks intended to ensure instance resolution termination.

IncoherentInstances

Since 6.8.1

Deprecated extension to weaken checks intended to ensure instance resolution termination.

In general, as discussed in *Instance resolution* (page 320), *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 324) and *IncoherentInstances* (page 324). 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 324).
- 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 324); 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 324); or if the instance is incoherent.

Now suppose that, in some client module, we are searching for an instance of the *target constraint* ($C \text{ ty1} \dots \text{ty}_n$). 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*.
- Eliminate any candidate IX for which both of the following hold:
 - There is another candidate IY that is strictly more specific; that is, IY is a substitution instance of IX but not vice versa.
 - Either IX is *overlappable*, or IY is *overlapping*. (This “either/or” design, rather than a “both/and” design, allow a client to deliberately override an instance from a library, without requiring a change to the library.)
- If exactly one non-incoherent candidate remains, select it. If all remaining candidates are incoherent, select an arbitrary one. Otherwise the search fails (i.e. when more than one surviving candidate is not incoherent).
- If the selected candidate (from the previous step) is incoherent, the search succeeds, returning that candidate.
- If not, find all instances 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, the search succeeds, returning the selected candidate; if not, 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 $(C \text{ Int Char})$ then only the first instance matches, and all is well. Similarly with $(C \text{ Bool Bool})$. But if we try to solve $(C \text{ Int Bool})$, 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)
```

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 324) 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`.

GHC is 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 instantiate 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 324) 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 can write this type signature yourself if you use the *FlexibleContexts* (page 312) 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 *FlexibleInstances* (page 321) to do this.)

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 324). 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.)

Instance signatures: type signatures in instance declarations

InstanceSigs

Since 7.6.1

Allow type signatures for members in instance definitions.

In Haskell, you can't write a type signature in an instance declaration, but it is sometimes convenient to do so, and the language extension *InstanceSigs* (page 327) allows you to do so. For example:

```
data T a = MkT a a
instance Eq a => Eq (T a) where
    (==) :: T a -> T a -> Bool    -- The 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.

- 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.
- 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 370) (*Lexically scoped type variables* (page 370)), the `forall b` scopes over the definition of `foo`, and in particular over the type signature for `xs`.

9.8.4 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 328)) 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 `GHC.Exts`.

Haskell's defaulting mechanism ([Haskell Report, Section 4.3.4](#)) is extended to cover string literals, when *OverloadedStrings* (page 328) 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 GHC.Exts( 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.

9.8.5 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 329) 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 328)), but with an additional type parameter that makes the text of the label available as a type-level string (see [Type-Level Literals](#) (page 363)). 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 375)) 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 256) 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 241) extension makes use of postfix hash signs; if [OverloadedLabels](#) (page 329) and [MagicHash](#) (page 241) are both enabled then `x#y` means `x# y`, but if only [OverloadedLabels](#) (page 329) is enabled then it means `x #y`. The [Unboxed-Tuples](#) (page 238) extension makes `#` a single lexeme, so when [UnboxedTuples](#) (page 238) 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 329).

When using [OverloadedLabels](#) (page 329) (or other extensions that make use of hash signs) in a `.hsc` file (see [Writing Haskell interfaces to C code: hsc2hs](#) (page 491)), 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 363) 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)
```


9.8.6 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 243).)

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]
```

```
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
```

Rebindable syntax

When desugaring list notation with *OverloadedLists* (page 331) 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 256), 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 256).

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])
```

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).

9.8.7 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 333) 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 (Trac #10318)

```
class (Frac (Frac a) ~ Frac a,
      Fractional (Frac a),
      IntegralDomain (Frac a))
  => IntegralDomain a where
  type Frac a :: *
```

Here the superclass cycle does terminate but it's not entirely straightforward to see that it does.

With the language extension [UndecidableSuperClasses](#) (page 333) GHC lifts all restrictions on superclass constraints. If there really is a loop, GHC will only expand it to finite depth.

9.9 Type families

TypeFamilies

Implies [MonoLocalBinds](#) (page 374), [KindSignatures](#) (page 369), [Explicit-Namespaces](#) (page 262)

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 [\[AssocDataTypes2005\]](#) (page 521) and associated type synonyms [\[AssocTypeSyn2005\]](#) (page 521). Type families themselves are described in Schrijvers 2008 [\[TypeFamilies2008\]](#) (page 521). 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 334). Additional information on the use of type families in GHC is available on [the Haskell wiki page on type families](#).

9.9.1 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 :: * -> *
```

The special family distinguishes family from standard data declarations. The result kind annotation is optional and, as usual, defaults to `*` 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 275) named arguments are entirely optional, so that we can declare `Array` alternatively with

```
data family Array :: * -> *
```

Unlike with ordinary data definitions, the result kind of a data family does not need to be `*`: it can alternatively be a kind variable (with *PolyKinds* (page 352)). Data instances' kinds must end in `*`, however.

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 the flag *-Wunused-type-patterns* (page 86) 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 387). However, there are some differences. No error messages reporting the inferred types are generated, nor does the extension *PartialTypeSignatures* (page 387) have any effect.

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.

9.9.2 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 :: *
```

The special family distinguishes family from standard type declarations. The result kind annotation is optional and, as usual, defaults to `*` 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 :: * -> *   -- F's arity is 2,
                               -- although its overall kind is * -> * -> * -> *
```

Given this declaration the following are examples of well-formed and malformed types:

```
F Char [Int]      -- OK! Kind: * -> *
F Char [Int] Bool -- OK! Kind: *
F IO Bool         -- WRONG: kind mismatch in the first argument
F Bool           -- WRONG: unsaturated application
```

The result kind annotation is optional and defaults to `*` (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.

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 86) flag is enabled. The same rules apply as for *Data instance declarations* (page 335).

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 forall) 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 as the equations for open type family instances.

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 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
  H Int  = Int
  H Bool = Bool
  H a    = String
type instance H Char = Char    -- WRONG: cannot have instances of closed family
type family K a where
  -- OK!
```



```

type family G a b :: * -> *
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 `*` while the second uses `* -> *`.

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`.

However see *Type, class and other declarations* (page 25) for the overlap rules in GHCi.

Decidability of type synonym instances

UndecidableInstances

Relax restrictions on the decidability of type synonym family 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 322) is passed to the compiler, the above restrictions are not enforced and it is on the programmer to ensure termination of the normalisation of type families during type inference.

9.9.3 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 :: *
data instance F Int _ = Int
-- Equivalent to data instance F Int b = Int

type family T a :: *
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 388)).

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 389)).

9.9.4 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 :: * -> *
  ...

class Collects ce where
  type Elem ce :: *
  ...
```

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 them 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 :: *
```

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 have precisely the same as type given in the instance head. For example:

```
class Collects ce where
  type Elem ce :: *

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
  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 -> *
-- k is implicitly bound by an invisible kind pattern
newtype instance Nat :: (k -> *) -> (k -> *) -> * where
  Nat :: (forall xx. f xx -> g xx) -> Nat f g

class Funct f where
  type Codomain f :: *
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 -> *)
```

- 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 343)), 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 *).

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 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.
- 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 :: *) where
  type F1 a :: *
  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

  type F4 a
  type F4 b = a                -- BAD; 'a' is not in scope in the RHS

  type F5 a :: [k]
  type F5 a = ('[] :: [x])    -- OK; the kind variable x is implicitly
                                -- bound by an invisible kind pattern
                                -- on the LHS

  type F6 a
  type F6 a =
    Proxy ('[] :: [x])        -- BAD; the kind variable x is not bound,
                                -- even by an invisible kind pattern

  type F7 (x :: a) :: [a]
```

```
type F7 x = ('[] :: [a])      -- OK; the kind variable a is implicitly
                               bound by the kind signature of the
                               LHS type pattern
```

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 352) 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 287) clause (at the top level).

9.9.5 Import and export

The rules for export lists (Haskell Report [Section 5.2](#)) needs 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 :: * -> *
  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:

```
or      module Y( D(..) ) where ...
       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.

9.9.6 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 320). 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 ...
```

9.9.7 Injective type families

TypeFamilyDependencies

Implies [TypeFamilies](#) (page 334)

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 Haskell Symposium 2015 paper [Injective type families for Haskell](#).

Syntax of injectivity annotation

Injectivity annotation is added after type family head and consists of two parts:

- a type variable that names the result of a type family. Syntax: `= tyvar` or `= (tyvar :: kind)`. 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 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 injectivity annotation against type family equations

Once the user declares type family to be injective GHC must verify that this declaration is correct, ie. 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 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.
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 preceeding 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.

9.10 Datatype promotion

DataKinds

Since 7.4.1

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 349), and described in more detail in the paper [Giving Haskell a Promotion](#), which appeared at TLDI 2012.

9.10.1 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 `*`) and type constructors (e.g. kind `* -> * -> *`). In particular when using advanced type system features, such as type families ([Type families](#) (page 334)) or GADTs ([Generalised Algebraic Data Types \(GADTs\)](#) (page 275)), 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 :: * -> * -> * where
  Nil  :: Vec a Ze
  Cons :: a -> Vec a n -> Vec a (Su n)
```

The kind of `Vec` is `* -> * -> *`. 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 349), the example above can then be rewritten to:

```
data Nat = Ze | Su Nat

data Vec :: * -> Nat -> * 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.

9.10.2 Overview

With *DataKinds* (page 349), 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 = Pair a b

data Sum a b = L a | R b
```

give rise to the following kinds and type constructors (where promoted constructors are prefixed by a tick '):

```
Nat :: *
'Zero :: Nat
'Succ :: Nat -> Nat

List :: * -> *
'Nil  :: forall k. List k
'Cons :: forall k. k -> List k -> List k

Pair  :: * -> * -> *
'Pair :: forall k1 k2. k1 -> k2 -> Pair k1 k2

Sum  :: * -> * -> *
'L   :: k1 -> Sum k1 k2
'R   :: k2 -> Sum k1 k2
```

The following restrictions apply to promotion:

- We promote data types and newtypes; type synonyms and type/data families are not promoted (*Type families* (page 334)).
- We only promote types whose kinds are of the form `* -> ... -> * -> *`. In particular, we do not promote higher-kinded datatypes such as `data Fix f = In (f (Fix f))`, or datatypes whose kinds involve promoted types such as `Vec :: * -> Nat -> *`.
- We do not promote data constructors that are kind polymorphic, involve constraints, mention type or data families, or involve types that are not promotable.

The flag *TypeInType* (page 352) (which implies *DataKinds* (page 349)) relaxes some of these restrictions, allowing:

- Promotion of type synonyms and type families, but not data families. GHC's type theory just isn't up to the task of promoting data families, which requires full dependent types.
- All datatypes, even those with rich kinds, get promoted. For example:

```
data Proxy a = Proxy
data App f a = MkApp (f a)  -- App :: forall k. (k -> *) -> k -> *
x = Proxy :: Proxy ('MkApp ('Just 'True))
```

9.10.3 Distinguishing between types and constructors

In the examples above, all promoted constructors are prefixed with a single quote mark '. This mark tells GHC to look in the data constructor namespace for a name, not the type

(constructor) namespace. Consider

```
data P = MkP      -- 1
data Prom = P     -- 2
```

We can thus distinguish the type `P` (which has a constructor `MkP`) from the promoted data constructor `'P` (of kind `Prom`).

As a convenience, GHC allows you to omit the quote mark when the name is unambiguous. However, our experience has shown that the quote mark helps to make code more readable and less error-prone. GHC thus supports *-Wunticked-promoted-constructors* (page 84) that will warn you if you use a promoted data constructor without a preceding quote mark.

Just as in the case of Template Haskell (*Syntax* (page 394)), 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`
```

9.10.4 Promoted list and tuple types

With *DataKinds* (page 349), 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 :: [*] -> * where
  HNil    :: HList '[]
  HCons   :: a -> HList t -> HList (a ': t)

data Tuple :: (*,*) -> * 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 265) because of infix type operator `(':')`

9.10.5 Promoting existential data constructors

Note that we do promote existential data constructors that are otherwise suitable. For example, consider the following:

```
data Ex :: * 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 :: *} (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 [Trac #7347](#).

9.11 Kind polymorphism and Type-in-Type

TypeInType

Implies [PolyKinds](#) (page 352), [DataKinds](#) (page 349), [KindSignatures](#) (page 369)

Since 8.0.1

Allow kinds to be as intricate as types, allowing explicit quantification over kind variables, higher-rank kinds, and the use of type synonyms and families in kinds, among other features.

PolyKinds

Implies [KindSignatures](#) (page 369)

Since 7.4.1

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.

9.11.1 The difference between TypeInType and PolyKinds

It is natural to consider [TypeInType](#) (page 352) as an extension of [PolyKinds](#) (page 352). The latter simply enables fewer features of GHC's rich kind system than does the former. The need for two separate extensions stems from their history: [PolyKinds](#) (page 352) was introduced

for GHC 7.4, when it was experimental and temperamental. The wrinkles were smoothed out for GHC 7.6. [TypeInType](#) (page 352) was introduced for GHC 8.0, and is currently experimental and temperamental, with the wrinkles to be smoothed out in due course. The intent of having the two extensions is that users can rely on [PolyKinds](#) (page 352) to work properly while being duly sceptical of [TypeInType](#) (page 352). In particular, we recommend enabling [-dcore-lint](#) (page 202) whenever using [TypeInType](#) (page 352); that extension turns on a set of internal checks within GHC that will discover bugs in the implementation of [TypeInType](#) (page 352). Please report bugs at [our bug tracker](#).

Although we have tried to allow the new behavior only when [TypeInType](#) (page 352) is enabled, some particularly thorny cases may have slipped through. It is thus possible that some construct is available in GHC 8.0 with [PolyKinds](#) (page 352) that was not possible in GHC 7.x. If you spot such a case, you are welcome to submit that as a bug as well. We flag newly available capabilities below.

9.11.2 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 `(* -> *) -> * -> *`. But this is overly specific, because another suitable Haskell 98 kind for `App` is `((* -> *) -> *) -> (* -> *) -> *`, where the kind assigned to `a` is `* -> *`. Indeed, without kind signatures ([KindSignatures](#) (page 369)), it is necessary to use a dummy constructor to get a Haskell compiler to infer the second kind. With kind polymorphism ([PolyKinds](#) (page 352)), GHC infers the kind for all `k`. `(k -> *) -> k -> *` 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 *
data T a = MkT (a Int)    -- `a` is inferred to have kind (* -> *)
App T Maybe      -- `k` is instantiated to (* -> *)
```

9.11.3 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 `*` is just `*`. It is true that the `* :: *` 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.

9.11.4 Principles of kind inference

Generally speaking, when *PolyKinds* (page 352) is on, GHC tries to infer the most general kind for a declaration. In this case 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 `*` and we get `F :: * -> *`. You can still declare F to be kind-polymorphic using kind signatures:

```
type family F1 a           -- F1 :: * -> *
type family F2 (a :: k)    -- F2 :: forall k. k -> *
type family F3 a :: k      -- F3 :: forall k. * -> 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 “*”, except when directed otherwise by a kind signature. Examples: data and open type family declarations.

This rule has occasionally-surprising consequences (see [Trac #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
  type F1 a               -- declarations, but the class forces (a :: k)
                          -- so D1, F1 :: forall k. k -> *

data D2 a                -- No right-hand side so D2 :: * -> *
type F2 a                -- No right-hand side so F2 :: * -> *
```

The kind-polymorphism from the class declaration makes D1 kind-polymorphic, but not so D2; and similarly F1, F1.

9.11.5 Complete user-supplied kind signatures and polymorphic recursion

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 :: (* -> *) -> * -> *
```

The recursive use of `T` forced the second argument to have kind `*`. 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 -> *) :: k -> * 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 `*`.

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 -> *) -> k -> *      where ...
-- Yes; T1 :: forall k. (k->*) -> k -> *

data T2 (a :: k -> *) :: k -> *    where ...
-- Yes; T2 :: forall k. (k->*) -> k -> *

data T3 (a :: k -> *) (b :: k) :: * where ...
-- Yes; T3 :: forall k. (k->*) -> k -> *

data T4 (a :: k -> *) (b :: k)      where ...
-- Yes; T4 :: forall k. (k->*) -> k -> *

data T5 a (b :: k) :: *            where ...
-- No; kind is inferred

data T6 a b                        where ...
-- No; kind is inferred
```

- For a datatype with a top-level `::` when *TypeInType* (page 352) is in effect: all kind variables introduced after the `::` must be explicitly quantified.

```
-- -XTypeInType is on
data T1 :: k -> *          -- No CUSK: `k` is not explicitly quantified
data T2 :: forall k. k -> * -- CUSK: `k` is bound explicitly
data T3 :: forall (k :: *). k -> * -- still a CUSK
```

Note that the first example would indeed have a CUSK without *TypeInType* (page 352).

- 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 `*`:

```
data family D1 a           -- D1 :: * -> *
data family D2 (a :: k)    -- D2 :: forall k. k -> *
data family D3 (a :: k) :: * -- D3 :: forall k. k -> *
type family S1 a :: k -> * -- S1 :: forall k. * -> k -> *
```

- 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.

With `TypeInType` (page 352) enabled, 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 -> *
data X (a :: Proxy k)
```

According to the rules above `X` has a CUSK. Yet, what is the kind of `k`? It is impossible to know. This code is thus rejected as masquerading as having a CUSK, but not really. If you wish `k` to be polykinded, it is straightforward to specify this:

```
data X (a :: Proxy (k1 :: k2))
```

The above definition is indeed fully fixed, with no masquerade.

9.11.6 Kind inference in closed type families

Although all open type families are considered to have a complete user-supplied kind signature, 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, as 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 family F3 (a :: k) :: k where
  F3 True  = False
  F3 False = True
  F3 x     = x
-- OK
```

9.11.7 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 `*`. GHC could theoretically propagate this information back into the instance head, and make that instance declaration apply only to type of kind `*`, 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.

9.11.8 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 370)):

```
data Proxy a    -- Proxy :: forall k. k -> *
p :: forall a. Proxy a
p = Proxy :: Proxy (a :: *)
```

GHC reports an error, saying that the kind of `a` should be a kind variable `k`, not `*`. 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 `*`. The function definition is then rejected for being more specific than its type signature.

9.11.9 Explicit kind quantification

Enabled by [TypeInType](#) (page 352), GHC now supports explicit kind quantification, as in these examples:

```
data Proxy :: forall k. k -> *
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.

For backward compatibility, kind variables *do not* need to be bound explicitly, even if the type starts with `forall`.

Accordingly, the rule for kind quantification in higher-rank contexts has changed slightly. In GHC 7, if a kind variable was mentioned for the first time in the kind of a variable bound in a non-top-level `forall`, the kind variable was bound there, too. That is, in `f :: (forall (a :: k). ...) -> ...`, the `k` was bound by the same `forall` as the `a`. In GHC 8, however, all kind variables mentioned in a type are bound at the outermost level. If you want one bound in a higher-rank `forall`, include it explicitly.

9.11.10 Kind-indexed GADTs

Consider the 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 ~ (* -> *)` and `a ~ Maybe`. The definition for `G` requires that [TypeInType](#) (page 352) be in effect, but pattern-matching on `G` requires no extension beyond [GADTs](#) (page 275). 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 for the datatype ([Complete user-supplied kind signatures and polymorphic recursion](#) (page 354)).

9.11.11 Higher-rank kinds

In concert with [RankNTypes](#) (page 380), 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 HRef1 HRef1 = Just HRef1
```

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
  HRef1 :: 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.

9.11.12 Constraints in kinds

As kinds and types are the same, kinds can now (with [TypeInType](#) (page 352)) contain type constraints. Only equality constraints are currently supported, however. We expect this to extend to other constraints in the future.

Here is an example of a constrained kind:

```
type family IsTypeLit a where
  IsTypeLit Nat      = 'True
  IsTypeLit Symbol   = 'True
  IsTypeLit a        = 'False

data T :: forall a. (IsTypeLit a ~ 'True) => a -> * where
  MkNat    :: T 42
  MkSymbol :: T "Don't panic!"
```

The declarations above are accepted. However, if we add `MkOther :: T Int`, we get an error that the equality constraint is not satisfied; `Int` is not a type literal. Note that explicitly quantifying with `forall a` is not necessary here.

9.11.13 The kind `*`

The kind `*` classifies ordinary types. Without [TypeInType](#) (page 352), this identifier is always in scope when writing a kind. However, with [TypeInType](#) (page 352), a user may wish to use `*` in a type or a type operator `*` in a kind. To make this all more manageable, `*` becomes an (almost) ordinary name with [TypeInType](#) (page 352) enabled. So as not to cause naming collisions, it is not imported by default; you must import `Data.Kind` to get `*` (but only with [TypeInType](#) (page 352) enabled).

The only way `*` is unordinary is in its parsing. In order to be backward compatible, `*` is parsed as if it were an alphanumeric identifier; note that we do not write `Int :: (*)` but just plain `Int :: *`. Due to the bizarreness with which `*` is parsed—and the fact that it is the only such operator in GHC—there are some corner cases that are not handled. We are aware of three:

- In a Haskell-98-style data constructor, you must put parentheses around `*`, like this:

```
data Universe = Ty (*) | Num Int | ...
```

- In an import/export list, you must put parentheses around `*`, like this:

```
import Data.Kind ( type (*) )
```

Note that the keyword `type` there is just to disambiguate the import from a term-level `(*)`. (*Explicit namespaces in import/export* (page 262))

- In an instance declaration head (the part after the word `instance`), you must parenthesize `*`. This applies to all manners of instances, including the left-hand sides of individual equations of a closed type family.

The `Data.Kind` module also exports `Type` as a synonym for `*`. Now that type synonyms work in kinds, it is conceivable that we will deprecate `*` when there is a good migration story for everyone to use `Type`. If you like neither of these names, feel free to write your own synonym:

```
type Set = *    -- silly Agda programmers...
```

All the affordances for `*` also apply to `★`, the Unicode variant of `*`.

9.11.14 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.

9.11.15 Kind defaulting without PolyKinds

Without *PolyKinds* (page 352) or *TypeInType* (page 352) enabled, GHC refuses to generalise over kind variables. It thus defaults kind variables to `*` when possible; when this is not possible, an error is issued.

Here is an example of this in action:

```
{-# LANGUAGE TypeInType #-}
data Proxy a = P    -- inferred kind: Proxy :: k -> *
data Compose f g x = MkCompose (f (g x))
    -- inferred kind: Compose :: (b -> *) -> (a -> b) -> a -> *

-- 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 :: *) (b :: *) (f :: b -> *) (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 `*`, but `f` and `g` would be ill-kinded if defaulted. The definition for `z` is thus an error.

9.11.16 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 70) 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 71).

9.12 Levity 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 236)) like `Int#`. We thus have so-called levity polymorphism.

Here are the key definitions, all available from `GHC.Exts`:

```
TYPE :: RuntimeRep -> *    -- highly magical, built into GHC

data RuntimeRep = LiftedRep      -- for things like `Int`
                | UnliftedRep    -- for things like `Array#`
                | 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 * = TYPE LiftedRep    -- * 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.

9.12.1 No levity-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 levity polymorphic. We thus forbid such constructions, via the following straightforward rule:

No variable may have a levity-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 :: *) (b :: TYPE r).
      (a -> b) -> a -> b
f $ x = f x
```

Here, only b is levity polymorphic. There are no variables with a levity-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 levity-polymorphic.

9.12.2 Levity-polymorphic bottoms

We can use levity 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 levity-polymorphic variable, and so are accepted. Their polymorphism allows users to use these to conveniently stub out functions that return unboxed types.

9.12.3 Printing levity-polymorphic types

-fprint-explicit-runtime-rep

Print RuntimeRep parameters as they appear; otherwise, they are defaulted to 'LiftedRep'.

Most GHC users will not need to worry about levity polymorphism or unboxed types. For these users, seeing the levity 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 `*`.

Should you wish to see levity polymorphism in your types, enable the flag `-fprint-explicit-runtime-reps` (page 71).

9.13 Type-Level Literals

GHC supports numeric and string literals at the type level, giving convenient access to a large number of predefined type-level constants. Numeric literals are of kind `Nat`, while string literals are of kind `Symbol`. This feature is enabled by the [DataKinds](#) (page 349) language extension.

The kinds of the literals and all other low-level operations for this feature are defined in module `GHC.TypeLits`. Note that the module defines 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 262)).

Here is an example of using type-level numeric literals to provide a safe interface to a low-level function:

```
import GHC.TypeLits
import Data.Word
import Foreign

newtype ArrPtr (n :: Nat) a = ArrPtr (Ptr a)

clearPage :: ArrPtr 4096 Word8 -> IO ()
clearPage (ArrPtr p) = ...
```

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")
```

9.13.1 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 -> Integer
```

```
-- 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 :: Integer -> Maybe SomeNat
SomeNat    :: KnownNat n => Proxy n -> SomeNat
```

The operations on strings are similar.

9.13.2 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
```

9.14 Constraints in types

9.14.1 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.

9.14.2 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 71) is set.

9.14.3 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 71) is enabled.

9.14.4 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 442)). It is exported by `Data.Coerce`, which also contains the documentation. More details and discussion can be found in the paper “[Safe Coercions](#)”.

9.14.5 The Constraint kind

ConstraintKinds

Since 7.4.1

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 377) constraints, e.g. `?x::Int` (with the *ImplicitParams* (page 377) extension)
- *Equality constraints* (page 364), e.g. `a ~ Int` (with the *TypeFamilies* (page 334) or *GADTs* (page 275) extensions)

With the *ConstraintKinds* (page 366) 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 `GHC.Exts`). So for example type `Foo (f :: * -> Constraint) = forall b. f b => b -> b` is allowed, as well as examples involving type families:

```
type family Typ a b :: Constraint
type instance Typ Int b = Show b
type instance Typ Bool b = Num b

func :: Typ a b => a -> b -> b
func = ...
```

Note that because constraints are just handled as types of a particular kind, this extension 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 super-classes, but to do so you must use *UndecidableInstances* (page 322) to signal that you don't mind if the type checker fails to terminate.

9.15 Extensions to type signatures

9.15.1 Explicit universal quantification (forall)

ExplicitForAll

Since 6.12.1

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 367) 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 370)).

Notes:

- With *ExplicitForAll* (page 367), `forall` becomes a keyword; you can't use `forall` as a type variable any more!
- As well in type signatures, you can also use an explicit `forall` in an instance declaration:

```
instance forall a. Eq a => Eq [a] where ...
```

- If the *-Wunused-foralls* (page 86) 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`.

9.15.2 The context of a type signature

The *FlexibleContexts* (page 312) extension lifts the Haskell 98 restriction that the type-class constraints in a type signature must have the form *(class type-variable)* or *(class (type-variable type1 type2 ... typen))*. With *FlexibleContexts* (page 312) these type signatures are perfectly okay

```
g :: Eq [a] => ...
g :: Ord (T a ()) => ...
```

The flag *FlexibleContexts* (page 312) also lifts the corresponding restriction on class declarations (*The superclasses of a class declaration* (page 312)) and instance declarations (*Relaxed rules for instance contexts* (page 321)).

9.15.3 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 368) 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 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 375) 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 368) allows you to switch off the ambiguity check. However, even with ambiguity checking switched off, GHC will complain about a function that can *never* be called, such as this one:

```
f :: (Int ~ Bool) => a -> a
```

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

1. that each universally quantified type variable `tvi` must be “reachable” from type, and
2. 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.

9.15.4 Explicitly-kinded quantification

KindSignatures

Since 6.8.1

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 369).

This extension enables kind signatures in the following places:

- data declarations:

```
data Set (cxt :: * -> *) a = Set [a]
```

- type declarations:

```
type T (f :: * -> *) = f Int
```

- class declarations:

```
class (Eq a) => C (f :: * -> *) a where ...
```

- forall's in type signatures:

```
f :: forall (cxt :: * -> *) . Set cxt Int
```

The parentheses are required. Some of the spaces are required too, to separate the lexemes. If you write `(f :: *->*)` you will get a parse error, because `::*->*` is a single lexeme in Haskell.

As part of the same extension, you can put kind annotations in types as well. Thus:

```
f :: (Int :: *) -> Int
g :: forall a. a -> (a :: *)
```

The syntax is

```
atype ::= '(' ctype '::' kind ')'
```

The parentheses are required.

9.16 Lexically scoped type variables

ScopedTypeVariables

Implies *ExplicitForAll* (page 367)

Since 6.8.1

Enable lexical scoping of type variables explicitly introduced with `forall`.

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 371)). 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.

9.16.1 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 371))
- An expression type signature ([Expression type signatures](#) (page 372))
- A pattern type signature ([Pattern type signatures](#) (page 372))
- Class and instance declarations ([Class and instance declarations](#) (page 373))

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

<code>(e :: a -> a)</code>	means	<code>(e :: a -> a)</code>
<code>(e :: b -> b)</code>	means	<code>(e :: forall b. b->b)</code>
<code>(e :: a -> b)</code>	means	<code>(e :: forall b. a->b)</code>

9.16.2 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 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!
```

The binding for `f3` is a pattern binding, and so its type signature does not bring `a` into scope. However `f1` is a function binding, and `f2` binds a bare variable; in both cases the type signature brings `a` into scope.

9.16.3 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)`.

9.16.4 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*. This 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 *cannot* already be in scope, because it is bound by the pattern match. GHC's rule is that in this situation (and only then), a pattern type signature can mention a type variable that is not already in scope; the effect is to bring it into scope, standing for the existentially-bound type variable.

When a pattern type signature binds a type variable in this way, GHC insists that the type variable is bound to a *rigid*, or fully-known, type variable. This means that any user-written type signature always stands for a completely known type.

If all this seems a little odd, we think so too. 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.

This is (now) the *only* situation in which a pattern type signature is allowed to mention a lexical variable that is not already in scope. For example, both `f` and `g` would be illegal if `a` was not already in scope.

9.16.5 Class and instance declarations

The type variables in the head of a class or instance declaration scope over the methods defined in the `where` part. You do not even need an explicit `forall` (although you are allowed an explicit `forall` in an instance declaration; see [Explicit universal quantification \(forall\)](#) (page 367)). 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]]))
```

9.17 Bindings and generalisation

9.17.1 Switching off the dreaded 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 374). 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 55)).

9.17.2 Let-generalisation

MonoLocalBinds

Since 6.12.1

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 374) GHC implements a slightly more conservative policy, using the following rules:

- A variable is *closed* if and only if
 - the variable is `let`-bound
 - one of the following holds:
 - * the variable has an explicit type signature that has no free type variables, or
 - * its binding group is fully generalised (see next bullet)
- A binding group is *fully generalised* if and only if
 - each of its free variables is either imported or closed, and
 - the binding is not affected by the monomorphism restriction ([Haskell Report, Section 4.5.5](#))

For example, consider

```
f x = x + 1
g x = let h y = f y * 2
      k z = z + x
      in h x + k x
```

Here `f` is generalised because it has no free variables; and its binding group is unaffected by the monomorphism restriction; and hence `f` is closed. The same reasoning applies to `g`, except that it has one closed free variable, namely `f`. Similarly `h` is closed, *even though it is not bound at top level*, because its only free variable `f` is closed. But `k` is not closed, because it mentions `x` which is not closed (because it is not `let`-bound).

Notice that a top-level binding that is affected by the monomorphism restriction is not closed, and hence may in turn prevent generalisation of bindings that mention it.

The rationale for this more conservative strategy is given in [the papers](#) “Let should not be generalised” and “Modular type inference with local assumptions”, and a related [blog post](#).

The extension `MonoLocalBinds` (page 374) is implied by `TypeFamilies` (page 334) and `GADTs` (page 275). You can switch it off again with `NoMonoLocalBinds` but type inference becomes less predicatable if you do so. (Read the papers!)

9.17.3 Kind generalisation

Just as `MonoLocalBinds` (page 374) places limitations on when the *type* of a *term* is generalised (see [Let-generalisation](#) (page 374)), it also limits when the *kind* of a *type signature* is generalised. Here is an example involving [type signatures on instance declarations](#) (page 327):

```
data Proxy a = Proxy
newtype Tagged s b = Tagged b

class C b where
  c :: forall (s :: k). Tagged s b

instance C (Proxy a) where
  c :: forall s. Tagged s (Proxy a)
  c = Tagged Proxy
```

With `MonoLocalBinds` (page 374) enabled, this `C (Proxy a)` instance will fail to typecheck. The reason is that the type signature for `c` captures `a`, an outer-scoped type variable, which means the type signature is not closed. Therefore, the inferred kind for `s` will *not* be generalised, and as a result, it will fail to unify with the kind variable `k` which is specified in the declaration of `c`. This can be worked around by specifying an explicit kind variable for `s`, e.g.,

```
instance C (Proxy a) where
  c :: forall (s :: k). Tagged s (Proxy a)
  c = Tagged Proxy
```

or, alternatively:

```
instance C (Proxy a) where
  c :: forall k (s :: k). Tagged s (Proxy a)
  c = Tagged Proxy
```

This declarations are equivalent using Haskell’s implicit “add implicit forall” rules (see [Implicit quantification](#) (page 383)). The implicit forall rules are purely syntactic and are quite separate from the kind generalisation described here.

9.18 Visible type application

TypeApplications

Since 8.0.1

Allow the use of type application syntax.

The [TypeApplications](#) (page 375) 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.

Here are the details:

- If an identifier's type signature does not include an explicit `forall`, the type variable arguments appear in the left-to-right order in which the variables appear in the type. So, `foo :: Monad m => a b -> m (a c)` will have its type variables ordered as `m`, `a`, `b`, `c`.
- 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.

For example: if we have `bar :: Proxy (a :: (j, k)) -> b`, then the variables are ordered `j`, `k`, `a`, `b`.

- Visible type application is available to instantiate only user-specified type variables. This means that in `data Proxy a = Proxy`, the unmentioned kind variable used in `a`'s kind is *not* available for visible type application.
- 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 380) extension ([Lexically scoped type variables](#) (page 370)), 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 387)) 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, it is *not* necessary to specify [PartialTypeSignatures](#) (page 387) and your code will not generate a warning informing you of the omitted type.
- When printing types with [-fprint-explicit-foralls](#) (page 70) enabled, type variables not available for visible type application are printed in braces. We can observe this behavior in a GHCi session:

```
> :set -XTypeApplications -fprint-explicit-foralls
> let myLength1 :: Foldable f => f a -> Int; myLength1 = length
> :type +v myLength1
myLength1 :: forall (f :: * -> *) a. Foldable f => f a -> Int
> let myLength2 = length
> :type +v myLength2
myLength2 :: forall {a} {t :: * -> *}. Foldable t => t a -> Int
> :type +v 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 +v` (page 53) 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.

Also note the use of `:type +v` (page 53) in the GHCi session above instead of `:type` (page 53). This is because `:type` (page 53) gives you the type that would be inferred for a variable assigned to the expression provided (that is, the type of `x` in `let x = <expr>`). As we saw above with myLength2, this type will have no variables available to visible type application. On the other hand, `:type +v` (page 53) gives you the actual type of the expression provided. To illustrate this:

```
> :type myLength1
myLength1 :: forall {a} {f :: * -> *}. Foldable f => f a -> Int
> :type myLength2
myLength2 :: forall {a} {t :: * -> *}. Foldable t => t a -> Int
```

Using `:type` (page 53) might lead one to conclude that none of the type variables in myLength1's type signature are available for type application. This isn't true, however! Be sure to use `:type +v` (page 53) if you want the most accurate information with respect to visible type application properties.

9.19 Implicit parameters

ImplicitParams

Since 6.8.1

Allow definition of functions expecting implicit parameters.

Implicit parameters are implemented as described in [Lewis2000] (page 521) and enabled with the option `ImplicitParams` (page 377). (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 $(?x::t') \Rightarrow t$, 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 `sortBy` 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
```

9.19.1 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 $(?x :: a) \Rightarrow (a, a)$, and not $(?x :: a, ?x :: b) \Rightarrow (a, b)$, 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`.

9.19.2 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 `lets` 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
```

9.19.3 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.

9.19.4 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.

9.20 Arbitrary-rank polymorphism

RankNTypes

Implies [ExplicitForAll](#) (page 367)

Since 6.8.1

Allow types of arbitrary rank.

Rank2Types

Since 6.8.1

A deprecated alias of [RankNTypes](#) (page 380).

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

f4 :: Int -> (forall a. a -> a)
```

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 380) (which implies [ExplicitForAll](#) (page 367)) 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 (see `f4`, for example) 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 370))

The [RankNTypes](#) (page 380) option is also required for any type with a forall or context to the right of an arrow (e.g. `f :: Int -> forall a. a->a`, or `g :: Int -> Ord a => a -> a`). Such types are technically rank 1, but are clearly not Haskell-98, and an extra extension did not seem worth the bother.

In particular, in data and newtype declarations the constructor arguments may be polymorphic types of any rank; see examples in [Examples](#) (page 381). 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 384)).

The obsolete language options `PolymorphicComponents` and [Rank2Types](#) (page 380) are synonyms for [RankNTypes](#) (page 380). They used to specify finer distinctions that GHC no longer makes. (They should really elicit a deprecation warning, but they don't, purely to avoid the need to library authors to change their old flags specifications.)

9.20.1 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.

As for type signatures, implicit quantification happens for non-overloaded types too. So if you write this:

```
f :: (a -> a) -> a
```

it's just as if you had written this:

```
f :: forall a. (a -> a) -> a
```

That is, since the type variable `a` isn't in scope, it's implicitly universally quantified.

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 (\xy->x) 3

a2, a3 :: Swizzle
a2 = MkSwizzle sort
a3 = MkSwizzle reverse

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 -> [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.

9.20.2 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 forall 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 370)), 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.

9.20.3 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
    -- 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)
```

9.21 Impredicative polymorphism

ImpredicativeTypes

Implies [RankNTypes](#) (page 380)

Since 6.10.1

Allow impredicative polymorphic types.

In general, GHC will only instantiate a polymorphic function at a monomorphic type (one with no `forall`s). 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 extremely flaky support for *impredicative polymorphism*, enabled with [ImpredicativeTypes](#) (page 384). If it worked, this would mean that you *could* call a polymorphic function at a polymorphic type, and parameterise data structures over polymorphic types. For example:

```
f :: Maybe (forall a. [a] -> [a]) -> Maybe ([Int], [Char])
f (Just g) = Just (g [3], g "hello")
f Nothing = Nothing
```

Notice here that the `Maybe` type is parameterised by the *polymorphic* type `(forall a. [a] -> [a])`. However *the extension should be considered highly experimental, and certainly un-supported*. You are welcome to try it, but please don’t rely on it working consistently, or working the same in subsequent releases. See [this wiki page](#) for more details.

If you want impredicative polymorphism, the main workaround is to use a newtype wrapper. The `id runST` example can be written using this workaround like this:

```
runST :: (forall s. ST s a) -> a
id :: forall b. b -> b

newtype Wrap a = Wrap { unwrap :: (forall s. ST s a) -> a }

foo :: (forall s. ST s a) -> a
foo = unwrap (id (Wrap runST))
    -- Here id is called at monomorphic type (Wrap a)
```

9.22 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 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
  Relevant bindings include
    f :: a -> a (bound at hole.hs:2:1)
    x :: a (bound at hole.hs:2:3)
  In the expression:
  In an equation for `f': f x = _
```

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 76) 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 76).

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 77) 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 77).

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 76) option, which implies *-fdefer-typed-holes* (page 76) and *-fdefer-out-of-scope-variables* (page 77). See *Deferring type errors to runtime* (page 393).

- 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:5:15: error:
  Found hole: _x :: Bool
  Relevant bindings include
    p :: Bool (bound at Foo.hs:3:6)
    cons :: Bool -> [Bool] (bound at Foo.hs:3:1)

Foo.hs:5:20: 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
  Relevant bindings include cons :: [a] (bound 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

unbound.hs:1:13:
  Found hole '_x' with type: [a]
  Arising from: an undeclared identifier `_x' at unbound.hs:1:13-14
  Where: `a' is a rigid type variable bound by
         the inferred type of cons :: [a] at unbound.hs:1:1
  Relevant bindings include cons :: [a] (bound at unbound.hs:1:1)
  In the second argument of `(:)', namely `_x'
  In the expression: _x : _x
  In an equation for `cons': cons = _x : _x
```

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.)

There's a flag 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 the type signature for:
       f :: Eq a => a -> Bool
       at show_constraints.hs:3:1-22)
```

9.23 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 written with a leading underscore (e.g., `"_"`, `"_foo"`, `"_bar"`) called *wildcards*. Partial type signatures are to type signatures what [Typed Holes](#) (page 385) 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 385), 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 387) 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 77) 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.

9.23.1 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

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 390)), only named wildcards can occur in the constraints, e.g. the `_x` in `Show _x`.

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 389) extension should be enabled. The example below demonstrated the effect.

```
foo :: _a -> _a
foo _ = False
```

Compiling this program without enabling [NamedWildCards](#) (page 389) 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 389) (as well as [PartialTypeSignatures](#) (page 387)) 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 he knows or wants 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:
  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.

9.23.2 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 340).

Anonymous wildcards are also allowed in visible type applications (*Visible type application* (page 375)). 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.

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 394) 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 370) 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
```

9.24 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 349)) 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
```

9.25 Deferring type errors to runtime

While developing, sometimes it is desirable to allow compilation to succeed even if there are type errors in the code. Consider the following case:

```
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](#).

9.25.1 Enabling deferring of type errors

The flag `-fdefer-type-errors` (page 76) 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-type-errors` to suppress these warnings.

This flag implies the `-fdefer-typed-holes` (page 76) and `-fdefer-out-of-scope-variables` (page 77) flags, which enables this behaviour for *typed holes* (page 385) and variables. Should you so wish, it is possible to enable `-fdefer-type-errors` (page 76) without enabling `-fdefer-typed-holes` (page 76) or `-fdefer-out-of-scope-variables` (page 77), by explicitly specifying `-fno-defer-typed-holes` (page 76) or `-fno-defer-out-of-scope-variables` (page 77) on the command-line after the `-fdefer-type-errors` (page 76) flag.

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`.

9.25.2 Deferred type errors in GHCi

The flag `-fdefer-type-errors` (page 76) 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:
  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
```

9.26 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 400)) 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](#).

9.26.1 Syntax

TemplateHaskell

Implies `TemplateHaskellQuotes` (page 395)

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 394) to switch these syntactic extensions on. Alternatively, the [TemplateHaskellQuotes](#) (page 395) extension can be used to enable the quotation subset of Template Haskell (i.e. without splice syntax). The [TemplateHaskellQuotes](#) (page 395) extension is considered safe under [Safe Haskell](#) (page 449) while [TemplateHaskell](#) (page 394) is not.

- A splice is written `$x`, where `x` is an identifier, or `$(...)`, where the `"..."` is an arbitrary expression. There must be no space between the `"$"` and the identifier or parenthesis. This use of `"$"` 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 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).

- A expression quotation is written in Oxford brackets, thus:
 - `[| ... |]`, or `[e| ... |]`, where the `"..."` is an expression; the quotation has type `Q Exp`.
 - `[d| ... |]`, where the `"..."` is a list of top-level declarations; the quotation has type `Q [Dec]`.
 - `[t| ... |]`, where the `"..."` is a type; the quotation has type `Q Type`.
 - `[p| ... |]`, where the `"..."` is a pattern; the quotation has type `Q Pat`.

See [Where can they occur?](#) (page 391) for using partial type signatures in quotations.

- A *typed* expression splice is written `$$x`, where `x` is an identifier, or `$$(...)`, where the `"..."` is an arbitrary expression.

A typed expression splice can occur in place of an expression; the spliced expression must have type `Q (TExp a)`

- 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 `Q (TExp a)`.

Values of type `TExp a` may be converted to values of type `Exp` using the function `unType :: TExp a -> Exp`.

- 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 402).

- 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 (sadly) cannot be quoted in 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. There is no current escape mechanism in this (unusual) situation.

- ''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 :: Int -> Q 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 `Q 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 :: t -> Q Exp
```

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 296) language extension. See *Deriving Lift instances* (page 296) 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

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 :: Q Pat
mkPat = [p| (x, y) |]

-- in another module:
foo :: (Char, String) -> String
foo $(mkPat) = x : z

bar :: Q 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 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

```
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`.
- Expression quotations accept most Haskell language constructs. However, there are some GHC-specific extensions which expression quotations currently do not support, including
 - Recursive do-statements (see [Trac #1262](#))
 - Type holes in typed splices (see [Trac #10945](#) and [Trac #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 `Q [Dec]`, not `[Q Dec]`. Typed expression splices and quotations are supported.)

9.26.2 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 66), `--interactive` (page 66), or file-at-a-time). There used to be a restriction to the former two, but that restriction has been lifted.

9.26.3 Viewing Template Haskell generated code

The flag `-ddump-splices` (page 197) 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 196) flag (see *Dumping out compiler intermediate structures* (page 196)). For each file using Template Haskell, this will show the output in a `.dump-splices` file.

The flag `-dth-dec-file={file}` (page 197) shows the expansions of all top-level TH declaration splices, both typed and untyped, in the file `M.th.hs` where `M` is the name of the module 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 196) with `-ddump-splices` (page 197) but it always generates a file instead of being coupled to `-ddump-to-file` (page 196). 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 197)

```
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={file}` (page 197)

```
-- TH_pragma.hs:(6,4)-(8,26): Splicing declarations
foo :: Int -> Int
foo x = (x + 1)
```

9.26.4 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 )

-- 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 :: [Format] -> Q 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 :: String -> Q 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
```

9.26.5 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 207).
2. Then compile it again with *-prof* (page 207), 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 150) flag when building with *-prof* (page 207) and Template Haskell is used, GHC will emit an error message.

The second option is to add the flag `-fexternal-interpreter` (page 58) (see [Running the interpreter in a separate process](#) (page 58)), 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 58) and it should just work. (this option is experimental in GHC 8.0.x, but it may become the default in future releases).

9.26.6 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 [Trac #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 394) for more information.

Warning: [QuasiQuotes](#) (page 402) introduces an unfortunate ambiguity with list comprehension syntax. Consider the following,

```
let x = [v | v <- [0..10]]
```

Without [QuasiQuotes](#) (page 402) this is parsed as a list comprehension. With [QuasiQuotes](#) (page 402) this is parsed as a quasi-quote; however, this parse will fail due to the lack of a closing `]`. See [Trac #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 Expr` 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)
  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
```

9.27 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 <http://www.haskell.org/arrows/> <<http://www.haskell.org/arrows/>>‘__.

With the [Arrows](#) (page 404) 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 $\langle \text{cmd} \rangle^0$ up to $\langle \text{cmd} \rangle^9$ 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 $\langle \text{calts} \rangle$ are like $\langle \text{alts} \rangle$ 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.

9.27.1 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 432)) 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.

9.27.2 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.

9.27.3 Defining your own control structures

As we've 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)|)
```

9.27.4 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
```

9.27.5 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.

9.27.6 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.

9.28 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 411)) makes pattern matching and `let` bindings stricter.
- Strict data types (*StrictData* (page 412)) makes constructor fields strict by default, on a per-module basis.

- Strict pattern (*Strict* (page 412)) 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.

9.28.1 Bang patterns

BangPatterns

Since 6.8.1

Allow use of bang pattern syntax.

GHC supports an extension of pattern matching called *bang patterns*, written `!pat`. Bang patterns are under consideration for Haskell Prime. The [Haskell prime feature description](#) contains more discussion and examples than the material below.

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. Bang patterns can be nested of course:

```
f2 (!x, y) = [x,y]
```

Here, `f2` is strict in `x` but not in `y`.

Note the following points:

- A bang only really has an effect if it precedes a variable or wild-card pattern:

```
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.

- A bang pattern is allowed in a `let` or `where` clause, and makes the binding strict. For example:

```
let !x = e in body
let !(p,q) = e in body
```

In both cases `e` is evaluated before starting to evaluate `body`.

However, *nested* bangs in a `let/where` pattern binding behave uniformly with all other forms of pattern matching. 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`.

See *Semantics of let bindings with bang patterns* (page 415) for the detailed semantics.

- A pattern with a bang at the outermost level is not allowed at the top level of a module.
- Bang patterns work in case expressions too, of course:

```
g5 x = let y = f x in body
g6 x = case f x of { y -> body }
g7 x = case f x of { !y -> body }
```

The functions `g5` and `g6` mean exactly the same thing. But `g7` evaluates `(f x)`, binds `y` to the result, and then evaluates `body`.

- 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 in favour of the latter. If you want to define `(!)` with bang-patterns enabled, you have to do so using prefix notation:

```
(!) f x = 3
```

9.28.2 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.

9.28.3 Strict-by-default pattern bindings

Strict

Implies *StrictData* (page 412)

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 412).

- **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.

- **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 414) below). So `p` and `q` may end up bound to `undefined`. See also [Dynamic semantics of bang patterns](#) (page 415) 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.

9.28.4 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.

9.28.5 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 10, 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 (t):

```
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`:

FORCE

Replace any binding `!p = e` with `v = case e of p -> (x1, ..., xn); (x1, ..., xn) = v` and replace body with `v seq body`, where `v` is fresh. This translation works fine if `p` is already a variable `x`, but can obviously be optimised by not introducing a fresh variable `v`.

SPLIT

Replace any binding `p = e`, where `p` is not a variable, with `v = e; x1 = case v of p -> x1; ...; xn = case v of p -> xn`, where `v` is fresh and `x1.. xn` are the bound variables of `p`. Again if `e` is a variable, this can be optimised by not introducing a fresh variable.

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

==> (FORCE)
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, Left y) = e in body

====> (FORCE)
let v = case e of (Just x, Left y) -> (x,y)
      (x,y) = v
in v `seq` body

====> (SPLIT)
let v = case e of (Just x, Left y) -> (x,y)
      x = case v of (x,y) -> x
      y = case v of (x,y) -> y
in v `seq` body

====> (inline seq, float x,y bindings inwards)
let v = case e of (Just x, Left y) -> (x,y)
in case v of v -> let x = case v of (x,y) -> x
                  y = case v of (x,y) -> y
                  in body

====> (fluff up v's pattern; this is a standard Core optimisation)
let v = case e of (Just x, Left y) -> (x,y)
in case v of v@(p,q) -> let x = case v of (x,y) -> x
                        y = case v of (x,y) -> y
                        in body

====> (case of known constructor)
let v = case e of (Just x, Left y) -> (x,y)
in case v of v@(p,q) -> let x = p
                        y = q
                        in body

====> (inline x,y, v)
case (case e of (Just x, Left y) -> (x,y) of
      (p,q) -> body[p/x, q/y])

====> (case of case)
case e of (Just x, Left y) -> body[p/x, q/y]
```

The final form is just what we want: a simple case expression.

Here is a recursive case

```
letrec xs :: [Int] -- Recursive
      !xs = factorial y : xs
in body

====> (FORCE)
letrec xs = factorial y : xs in xs `seq` body

====> (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

==> (FORCE)
    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

==> (FORCE)
    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

9.29 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 88) 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 91) option. The option `-fno-ignore-asserts` (page 91) 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.

9.30 Static pointers

StaticPointers

Since 7.10.1

Allow use of static pointer syntax.

The language extension `StaticPointers` (page 418) 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.

9.30.1 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 1
ref2 = static inc
ref3 = static (inc 1)
ref4 = static ((\x -> x + 1) (1 :: Int))
```



```
ref5 y = static (let x = 1 in x)
ref6 y = let x = 1 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
ref8 (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 48) command may use *Static-Pointers* (page 418) and static expressions, statements entered on the REPL may not. This is a limitation of GHCi; see [Trac #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.

9.30.2 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 :: 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 show                -- No Typeable instance for (Show a => a -> String)
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
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))
```

9.31 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. The layout rule applies in pragmas, so the closing `-#` should start in a column to the right of the opening `{-#`.

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 #-}`.

9.31.1 LANGUAGE pragma

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 420)).

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 67)).

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.

9.31.2 OPTIONS_GHC pragma

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 64) 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 420)).

9.31.3 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.

9.31.4 WARNING and DEPRECATED pragmas

The `WARNING` pragma allows you to attach an arbitrary warning to a particular function, class, or type. A `DEPRECATED` pragma lets you specify that a particular function, class, or type is deprecated. There are two 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 265)).

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, and (c) uses in an export list. The latter reduces spurious complaints within a library in which one module gathers together and re-exports the exports of several others.

You can suppress the warnings with the flag `-Wno-warnings-deprecations` (page 77).

9.31.5 MINIMAL pragma

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 (a) that lack a default method in the class declaration, and (b) whose name that does not start with an underscore (c.f. [-Wmissing-methods](#) (page 82), [Warnings and sanity-checking](#) (page 74)).

This warning can be turned off with the flag `-Wno-missing-methods` (page 82).

9.31.6 INLINE and NOINLINE pragmas

These pragmas control the inlining of function definitions.

INLINE pragma

GHC (with `-O` (page 88), 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 88).

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.

Syntactically, an `INLINE` pragma for a function can be put anywhere its type signature could be put.

`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 425)) and `INLINABLE` (*`INLINABLE` pragma* (page 424)) pragmas.

INLINABLE pragma

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 438)). 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 426)).
- 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.

`NOINLINE` pragma

The `NOINLINE` 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).

`CONLIKE` modifier

An `INLINE` or `NOINLINE` pragma may have a `CONLIKE` modifier, which affects matching in RULEs (only). See *How rules interact with `CONLIKE` pragmas* (page 435).

Phase control

Sometimes you want to control exactly when in GHC's pipeline the `INLINE` 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` you'll see the sequence of phase numbers for successive runs of the simplifier. In an `INLINE` 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:

	<i>-- Before phase 2</i>	<i>Phase 2 and later</i>
<code>{-# INLINE [2] f #-}</code>	<i>-- No</i>	<i>Yes</i>
<code>{-# INLINE [~2] f #-}</code>	<i>-- Yes</i>	<i>No</i>
<code>{-# NOINLINE [2] f #-}</code>	<i>-- No</i>	<i>Maybe</i>
<code>{-# NOINLINE [~2] f #-}</code>	<i>-- Maybe</i>	<i>No</i>
<code>{-# INLINE f #-}</code>	<i>-- Yes</i>	<i>Yes</i>
<code>{-# NOINLINE f #-}</code>	<i>-- No</i>	<i>No</i>

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` and `NOINLINE`, the phase number says when inlining is allowed at all.
- The `INLINE` 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 RULEs (*Rewrite rules* (page 432)).

9.31.7 LINE pragma

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.

9.31.8 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.

9.31.9 RULES pragma

The `RULES` pragma lets you specify rewrite rules. It is described in [Rewrite rules](#) (page 432).

9.31.10 SPECIALIZE pragma

(UK spelling also accepted.) For key overloaded functions, you can create extra versions (NB: more code space) 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 424)).
- A `SPECIALIZE` has the effect of generating (a) a specialised version of the function and (b) a rewrite rule (see [Rewrite rules](#) (page 432)) 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 425)) 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
{-# SPECIALIZE f :: Int -> b -> b #-}

g :: (Eq a, Ix b) => a -> b -> b
{-# SPECIALIZE g :: (Eq a) => a -> Int -> Int #-}

h :: Eq a => a -> a -> a
{-# SPECIALIZE 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.

SPECIALIZE INLINE

A SPECIALIZE pragma can optionally be followed with an INLINE or NOINLINE pragma, optionally followed by a phase, as described in *INLINE and NOINLINE pragmas* (page 422). 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 425)) to `SPECIALISE INLINE` pragma, just like on an `INLINE` 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.

SPECIALIZE for imported functions

Generally, you can only give a `SPECIALIZE` pragma for a function defined in the same module. However if a function `f` is given an `INLINABLE` pragma at its definition site, then it can subsequently be specialised by importing modules (see [INLINABLE pragma](#) (page 424)). 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 )

  data T = T1 | T2 deriving( Eq, Ord )
  {-# SPECIALISE lookup :: [(T,a)] -> T -> Maybe a

```

Here, `lookup` is declared `INLINABLE`, 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` pragma in every module.

Moreover you often don’t even need the `SPECIALIZE` pragma in the first place. When compiling a module `M`, GHC’s optimiser (when given the `-O` (page 88) 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` 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.

Obsolete SPECIALIZE syntax

In earlier versions of GHC, it was possible to provide your own specialised function for a given type:

```
{-# SPECIALIZE hammeredLookup :: [(Int, value)] -> Int -> value = intLookup #-}
```

This feature has been removed, as it is now subsumed by the RULES pragma (see [Specialisation](#) (page 437)).

9.31.11 SPECIALIZE instance pragma

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.

9.31.12 UNPACK pragma

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 rebored (this is done automatically by the compiler).

Unpacking constructor fields should only be used in conjunction with `-O` (page 88)¹, in order to expose unfoldings to the compiler so the reboring can be removed as often as possible. For example:

```
f :: T -> Float
f (T f1 f2) = f1 + f2
```

The compiler will avoid reboring f1 and f2 by inlining + on floats, but only when `-O` (page 88) is on.

Any single-constructor data is eligible for unpacking; for example

¹ in fact, UNPACK has no effect without `-O`, for technical reasons (see tick 5252)

```
data T = T {-# UNPACK #-} !(Int,Int)
```

will store the two Ints 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.

See also the *-funbox-strict-fields* (page 97) flag, which essentially has the effect of adding *{-# UNPACK #-}* to every strict constructor field.

9.31.13 NOUNPACK pragma

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 97) and *-O* (page 88), the field of the constructor T is not unpacked.

9.31.14 SOURCE pragma

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 152).

9.31.15 COMPLETE pragmas

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 305). 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.

When specifying a COMPLETE pragma, the result types of all patterns must be consistent with each other. This is a sanity check as it would be impossible to match on all the patterns if the types were inconsistent.

The result type must also be unambiguous. Usually this can be inferred but when all the pattern synonyms in a group are polymorphic in the constructor the user must provide a type signature.

```
class LL f where
  go :: f a -> ()

instance LL [] where
  go _ = ()

pattern T :: LL f => f a
pattern T <- (go -> ())

{-# COMPLETE T :: [] #-}

-- No warning
foo :: [a] -> Int
foo T = 5
```

9.31.16 Disambiguating between multiple COMPLETE pragmas

What should happen if there are multiple COMPLETE sets that apply to a single set of patterns? Consider this example:

```
data T = MkT1 | MkT2 | MkT2Internal
{-# COMPLETE MkT1, MkT2 #-}
{-# COMPLETE MkT1, MkT2Internal #-}

f :: T -> Bool
f MkT1 = True
f MkT2 = False
```

Which COMPLETE pragma should be used when checking the coverage of the patterns in `f`? If we pick the COMPLETE set that covers `MkT1` and `MkT2`, then `f` is exhaustive, but if we pick the other COMPLETE set that covers `MkT1` and `MkT2Internal`, then `f` is *not* exhaustive, since it fails to match `MkT2Internal`. An intuitive way to solve this dilemma is to recognize that picking

the former COMPLETE set produces the fewest number of uncovered pattern clauses, and thus is the better choice.

GHC disambiguates between multiple COMPLETE sets based on this rationale. To make things more formal, when the pattern-match checker requests a set of constructors for some data type constructor `T`, the checker returns:

- The original set of data constructors for `T`
- Any COMPLETE sets of type `T`

GHC then checks for pattern coverage using each of these sets. If any of these sets passes the pattern coverage checker with no warnings, then we are done. If each set produces at least one warning, then GHC must pick one of the sets of warnings depending on how good the results are. The results are prioritized in this order:

1. Fewest uncovered clauses
2. Fewest redundant clauses
3. Fewest inaccessible clauses
4. Whether the match comes from the original set of data constructors or from a COMPLETE pragma (prioritizing the former over the latter)

9.31.17 OVERLAPPING, OVERLAPPABLE, OVERLAPS, and INCOHERENT pragmas

The pragmas `OVERLAPPING`, `OVERLAPPABLE`, `OVERLAPS`, `INCOHERENT` are used to specify the overlap behavior for individual instances, as described in Section [Overlapping instances](#) (page 324). The pragmas are written immediately after the instance keyword, like this:

```
instance {-# OVERLAPPING #-} C t where ...
```

9.32 Rewrite rules

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 198) to see what rules fired. If you need more information, then `-ddump-rule-firings` (page 198) shows you each individual rule firing and `-ddump-rule-rewrites` (page 198) 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.

9.32.1 Syntax

From a syntactic point of view:

- There may be zero or more rules in a RULES 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 425)), 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 475)), by making it possible to define a RULE that is never run by GHC, but is nevertheless parsed, type-checked etc, so that it is available to the plugin.

- Each 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.

- 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 RULE “forall” is treated as a keyword, regardless of any other flag settings. Furthermore, inside a RULE, the language extension *ScopedTypeVariables* (page 370) is automatically enabled; see *Lexically scoped type variables* (page 370).
- Like other pragmas, RULE pragmas are always checked for scope errors, and are type-checked. 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 432) flag is on (see *Semantics* (page 434)).

9.32.2 Semantics

From a semantic point of view:

- Rules are enabled (that is, used during optimisation) by the *-fenable-rewrite-rules* (page 432) flag. This flag is implied by *-O* (page 88), and may be switched off (as usual) by *-fno-enable-rewrite-rules* (page 432). (NB: enabling *-fenable-rewrite-rules* (page 432) without *-O* (page 88) may not do what you expect, though, because without *-O* (page 88) GHC ignores all optimisation information in interface files; see *-fignore-interface-pragmas* (page 91)). Note that *-fenable-rewrite-rules* (page 432) 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 437) 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.

9.32.3 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 RULE might fire.

The way to get predictable behaviour is to use a `NOINLINE` pragma, or an `INLINE[⟨phase⟩]` pragma, on `f`, to ensure that it is not inlined until its RULEs have had a chance to fire. The warning flag `-Winline-rule-shadowing` (page 87) (see [Warnings and sanity-checking](#) (page 74)) warns about this situation.

9.32.4 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` pragma is a modifier to `INLINE/NOINLINE` 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.

9.32.5 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 435).

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 x y = 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`.

9.32.6 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.

9.32.7 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

```
{-# 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` *when-ever 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.

9.32.8 Controlling what's going on in rewrite rules

- Use `-ddump-rules` (page 198) 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 198) to see what rules are being fired. If you add `-dppr-debug` (page 196) you get a more detailed listing.
- Use `-ddump-rule-firings` (page 198) or `-ddump-rule-rewrites` (page 198) to see in great detail what rules are being fired. If you add `-dppr-debug` (page 196) 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`! 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`.

9.33 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.

9.34 Generic classes

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](#) (page 439).

9.35 Generic programming

Using a combination of [DeriveGeneric](#) (page 440), [DefaultSignatures](#) (page 314), and [DeriveAnyClass](#) (page 302), you can easily do datatype-generic programming using the `GHC.Generics` framework. 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.

9.35.1 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 *
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 :: * -> *
  -- 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 -> *) where
  type Rep1 f :: k -> *

  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 `* -> *` by default, but if the [PolyKinds](#) (page 352) extension is enabled, then it can range of types of kind `k -> *`, for any kind `k`.

DeriveGeneric

Since 7.2.1

Allow automatic deriving of instances for the `Generic` typeclass.

Instances of these classes can be derived by GHC with the [DeriveGeneric](#) (page 440) 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 287) can also be used.

9.35.2 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 = 0 | 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) = 0 : 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 `[0, 0]`, `b` would be `[0, I]`, and `c` would be `[I]`. However, if GHC chooses `a :+: (b :+: c)`, then the encoding for `a` would be `[0]`, `b` would be `[I, 0]`, 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.

9.35.3 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`.

9.35.4 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 314) 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 302), 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.

9.35.5 More information

For more details please refer to the [Haskell Wiki page](#) or the original paper *[Generics2010]* (page 521).

9.36 Roles

Using *GeneralizedNewtypeDeriving* (page 297) (*Generalising the deriving clause* (page 297)), 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.

9.36.1 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 `Simple Age` and `Simple Int` are *not* the same. Lastly, `Phant Age` and `Phant Bool` have the same representation, even though `Age` and `Bool` are unrelated.

9.36.2 Role inference

What role should a given type parameter should 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`.

9.36.3 Role annotations

RoleAnnotations

Since 7.8.1

Allow role annotation syntax.

Sometimes the programmer wants to constrain the inference process. For example, the base library contains the following definition:

```
data Ptr a = Ptr Addr#
```

The idea is that `a` should really be a representational parameter, but role inference assigns it to phantom. This makes some level of sense: a pointer to an `Int` really is representationally the same as a pointer to a `Bool`. But, that's not at all how we want to use `Ptr`s! So, we want to be able to say

```
type role Ptr representational
data Ptr a = Ptr Addr#
```

The `type role` (enabled with [RoleAnnotations](#) (page 444)) declaration forces the parameter `a` to be at role `representational`, not role `phantom`. GHC then checks the user-supplied roles to make sure they don't break any promises. It would be bad, for example, if the user could make `BadIdea`'s role be `representational`.

As another 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 324) 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 152)), 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
```

9.37 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. For example, we can define

```
errorWithCallStack :: HasCallStack => String -> a
```

as a variant of `error` that will get its call-site (as of GHC 8.0, `error` already gets its call-site, but let's assume for the sake of demonstration that it does not). We can access the call-stack inside `errorWithCallStack` with `GHC.Stack.callStack`.

```
errorWithCallStack :: HasCallStack => String -> a
errorWithCallStack msg = error (msg ++ "\n" ++ prettyCallStack callStack)
```

Thus, if we call `errorWithCallStack` we will get a formatted call-stack alongside our error message.

```
ghci> errorWithCallStack "die"
*** Exception: die
CallStack (from HasCallStack):
  errorWithCallStack, called at <interactive>:2:1 in interactive:Ghci1
```

The `CallStack` will only extend as far as the types allow it, for example

```
myHead :: HasCallStack => [a] -> a
myHead [] = errorWithCallStack "empty"
myHead (x:xs) = x

bad :: Int
bad = myHead []
```

```
ghci> bad
*** Exception: empty
CallStack (from HasCallStack):
  errorWithCallStack, called at Bad.hs:8:15 in main:Bad
  myHead, called at Bad.hs:12:7 in main:Bad
```

includes the call-site of `errorWithCallStack` 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 `errorWithCallStack` line in the printed stack is not particularly enlightening, so we might choose to suppress it by freezing the `CallStack` that we pass to `errorWithCallStack`.

```
myHead :: HasCallStack => [a] -> a
myHead []      = withFrozenCallStack (errorWithCallStack "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.

9.37.1 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.

9.38 Concurrent and Parallel Haskell

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.

9.38.1 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](#).

9.38.2 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.

9.38.3 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 189), and run it with the RTS `-N <x>` (page 100) option; see [Using SMP parallelism](#) (page 99)). 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 100) RTS option.

9.38.4 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 448)), 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 + 1))
                    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 134).

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.

9.38.5 Data Parallel Haskell

GHC includes experimental support for Data Parallel Haskell (DPH). This code is highly unstable and is only provided as a technology preview. More information can be found on the corresponding [DPH wiki page](#).

9.39 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 458) for more details of this.

The design of Safe Haskell covers the following aspects:

- A [safe language](#) (page 452) 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 185) 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 459).

9.39.1 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

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`.

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:

```
-- 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 Monad RIO where
    return = UnsafeRIO . return
    (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 458) or [Trustworthy](#) (page 458) flag and compiling `Danger` with the [Safe](#) (page 458) flag. We explain each below.

The use of [Safe](#) (page 458) to compile `Danger` restricts the features of Haskell that can be used to a [safe subset](#) (page 452). This includes disallowing `unsafePerformIO`, Template Haskell, pure FFI functions, `RULES` and restricting the operation of Overlapping Instances. The [Safe](#) (page 458) 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 458), where GHC provides a mechanical guarantee that the code is safe. Or those modules compiled with [Trustworthy](#) (page 458), where the module author claims that the module is Safe.

This is why the `RIO` module is compiled with [Safe](#) (page 458) or [Trustworthy](#) (page 458)>, to allow the `Danger` module to import it. The [Trustworthy](#) (page 458) flag doesn't place any restrictions on the module like [Safe](#) (page 458) does (expect to restrict overlapping instances to [safe overlapping instances](#) (page 453)). 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 458) is a problem as an arbitrary module can use it to mark themselves as trusted, yet [Trustworthy](#) (page 458) doesn't offer any guarantees about the module, unlike [Safe](#) (page 458). To control the use of trustworthy modules it is recommended to use the [-fpackage-trust](#) (page 459) 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 458) 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 458), 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 455).

In the example, `Danger` can import module `RIO` because `RIO` is compiled with [Safe](#) (page 458). 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.

9.39.2 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 324), 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` — 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` — 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 458) are dropped. Rules defined in Trustworthy modules that M imports are still valid and will fire as usual.
- `OverlappingInstances` — 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 453) for details.
- `GeneralisedNewtypeDeriving` — GND is not allowed in the safe language. This is due to the ability of it 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 440) extension) are still allowed. Note that the only allowed [deriving strategy](#) (page 304) 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 439) section for more details.

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:

```
{-# LANGUAGE Safe #-}
module Class (TC(..)) where
  class TC a where { op :: a -> String }

{-# LANGUAGE Safe #-}
module Dangerous (TC(..)) where
  import Class

  instance
    {-# OVERLAPS #-}
    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 458) 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 458) 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 458) or *Trustworthy* (page 458), then compilation will fail. For *Unsafe* (page 459), no restriction is applied, and for modules using safe inference, they will be inferred unsafe.

9.39.3 Safe Imports

Safe Haskell enables a small extension to the usual import syntax of Haskell, adding a `safe` keyword:

```
import decl -> 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.

9.39.4 Trust and Safe Haskell Modes

Safe Haskell introduces the following three language flags:

- *Safe* (page 458) — 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 458) for more details of this.
- *Trustworthy* (page 458) — 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 453). 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 459) — Marks the module being compiled as unsafe so that modules compiled using *Safe* (page 458) 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 458) for more details.

The procedure to check if a module is trusted or not depends on if the `-fpackage-trust` (page 459) flag is present. The check is similar in both cases with the `-fpackage-trust` (page 459) flag enabling an extra requirement for trustworthy modules to be regarded as trusted.

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 458)
 - All of `M`'s direct imports are trusted by `C`
- or all of these hold:
 - The module was compiled with *Trustworthy* (page 458)

- 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 458) and it will be trusted. To control this, there is an additional definition of package trust (enabled with the *-fpackage-trust* (page 459) 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*.

Trust check (*-fpackage-trust* enabled)

When the *-fpackage-trust* (page 459) 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 457).

When the *-fpackage-trust* (page 459) 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 458)
 - All of *M*'s direct imports are trusted by *C*
- *or* all of these hold:
 - The module was compiled with *Trustworthy* (page 458)
 - 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 458) compiled modules.

The reason there are two modes of checking trust is that the extra requirement enabled by *-fpackage-trust* (page 459) 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 459) by default and turning it into a flag makes Safe Haskell an opt-in extension rather than an always on feature.

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 458), 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 458), 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 458). `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 458) modules.

Trustworthy Requirements

Module authors using the *Trustworthy* (page 458) 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 means that symbols exported should respect type safety and referential transparency.

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 163).

9.39.5 Safe Haskell Inference

In the case where a module is compiled without one of [Safe](#) (page 458), [Trustworthy](#) (page 458) or [Unsafe](#) (page 459) 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 458) 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 458) 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 450) 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 458) 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.

9.39.6 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 458) 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 453). It also allows the use of the safe import keyword.

- *Module Trusted* — Yes.
- *Module Trusted* (*-fpackage-trust* (page 459) 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 458). 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.

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 three 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.

-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.

9.39.7 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 185) 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 455) and that the *-fpackage-trust* (page 459) flag (see *flag* (page 457)) 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 473) is forbidden, as that would allow bypassing Safe Haskell restrictions. See [Trac #10826](#) for details.

FOREIGN FUNCTION INTERFACE (FFI)

ForeignFunctionInterface

Since 6.8.1

Allow use of the Haskell foreign function interface.

GHC (mostly) conforms to the Haskell Foreign Function Interface, whose definition is part of the Haskell Report on <http://www.haskell.org/>.

FFI support is enabled by default, but can be enabled or disabled explicitly with the *ForeignFunctionInterface* (page 461) 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 462), 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.

10.1 GHC differences to the FFI Chapter

10.1.1 Guaranteed call safety

The FFI addendum stipulates that an implementation is free to implement an `unsafe` call by performing a `safe` call (and therefore may run in an arbitrary thread and may be subject to concurrent garbage collection). This greatly constrains library authors since it implies that it is never safe to pass any heap object reference to a foreign function, even if invoked with an `unsafe` call. For instance, it is often desirable to pass an unpinned `ByteArray#`s directly to native code to avoid making an otherwise-unnecessary copy. However, this can only be done safely under `unsafe` call semantics as otherwise the array may be moved by the garbage collector in the middle of the call.

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. [Trac #13730](#)).

However, since version 8.4 this is no longer the case: GHC **guarantees** that garbage collection will never occur during an `unsafe` call, even in the bytecode interpreter.

10.2 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.

10.2.1 Unboxed types

The following unboxed types may be used as basic foreign types (see FFI Chapter, Section 8.6): `Int#`, `Word#`, `Char#`, `Float#`, `Double#`, `Addr#`, `StablePtr#` `a`, `MutableByteArray#`, `ForeignObj#`, and `ByteArray#`.

10.2.2 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
```

10.2.3 Primitive imports

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` (by way of `unsafeCoerce#`) and the result type is allowed to be an unboxed tuple or the type `Any`.

This feature is not intended for use outside of the core libraries that come with GHC. For more details see the [GHC developer wiki](#).

10.2.4 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, 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`.

If the system call is successfully interrupted, it will return to Haskell whereupon the exception can be raised. Be especially careful when using `interruptible` that the caller of the foreign function is prepared to deal with the consequences of the call being interrupted; on Unix it is good practice to check for `EINTR` always, but on Windows it is not typically necessary to handle `ERROR_OPERATION_ABORTED`.

10.2.5 The CAPI calling convention

CApiFFI

Since 7.10.1

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 = ...
```

10.2.6 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.

10.3 Using the FFI with GHC

The following sections also give some hints and tips on the use of the foreign function interface in GHC.

10.3.1 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 149).

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"

#ifdef __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 `#ifdef __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 505).

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 189), 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 187)), and the `-rtsopts[={none|some|all}]` (page 190) link-time flag overrides this. However, `-rtsopts[={none|some|all}]` (page 190) has no effect when `-no-hs-main` (page 189) is in use (and the same goes for `-with-rtsopts={opts}` (page 190)). To set these options we have to call a GHC-specific API instead of `hs_init()`:

```
#include <stdio.h>
#include "HsFFI.h"

#ifdef __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;

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 465), 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 = 2;
    char *argv[] = { "+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.

10.3.2 Using header files

C functions are normally declared using prototypes in a C header file. Earlier versions of GHC (6.8.3 and earlier) #included 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 180) (`-fasm` (page 186)) 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.

10.3.3 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 are currently implemented using `malloc/free`, so while they might be a more convenient way to structure your memory allocation than using one of the other forms of allocation, they won't be any more efficient. We do plan to provide an improved-performance implementation of Pools in the future, however.

10.3.4 Multi-threading and the FFI

In order to use the FFI in a multi-threaded setting, you must use the `-threaded` (page 189) option (see *Options affecting linking* (page 187)).

Foreign imports and multi-threading

When you call a foreign imported function that is annotated as `safe` (the default), and the program was linked using `-threaded` (page 189), then the call will run concurrently with other running Haskell threads. If the program was linked without `-threaded` (page 189), then the other Haskell threads will be blocked until the call returns.

This means that if you need to make a foreign call to a function that takes a long time or blocks indefinitely, then you should mark it `safe` and use `-threaded` (page 189). Some library

functions make such calls internally; their documentation should indicate when this is the case.

If you are making foreign calls from multiple Haskell threads and using *-threaded* (page 189), make sure that the foreign code you are calling is thread-safe. In particular, 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 469)).

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 100)). The `-N (x)` (page 100) 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 462) 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 189), 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 497), `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.

10.3.5 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.

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.

11.1 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 be inspected and utilized when using GHC as a library or writing a compiler plugin.

11.1.1 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 399) 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 = ...
```

11.1.2 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 = ...
```

11.1.3 Annotating modules

You can annotate modules with the ANN pragma by using the module keyword. For example:

```
{-# ANN module (Just "A `Maybe String` annotation") #-}
```

11.2 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 DynFlags ( defaultLogAction )

main =
  defaultErrorHandler defaultLogAction $ 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](#).

11.3 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](#).

11.3.1 Using compiler plugins

Plugins can be specified on the command line with the `-fplugin={module}` (page 475) option where {module} is a module in a registered package that exports a plugin. Arguments can be given to plugins with the `-fplugin-opt={module}:{args}` (page 475) 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 475).

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 ...
$
```

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 475). 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 476) was not specified, then all other versions will become hidden. `-plugin-package <pkg>` (page 476) supports thinning and renaming described in *Thinning and renaming modules* (page 167).

Unlike `-package <pkg>` (page 165), 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 476), 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 476) flags to GHC. `-plugin-package-id <pkg-id>` (page 476) supports thinning and renaming described in *Thinning and renaming modules* (page 167).

-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 476) 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 166) flag); however, in the future there will be a separate field for specifying plugin dependencies specifically.

11.3.2 Writing compiler plugins

Plugins are modules that export at least a single identifier, `plugin`, of type `GhcPlugins.Plugin`. All plugins should import `GhcPlugins` 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, `GhcPlugins.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':

```
module DoNothing.Plugin (plugin) where
import GhcPlugins

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'.

11.3.3 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, vectorisation, 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 `CoreToDos` 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:

```
install :: [CommandLineOption] -> [CoreToDo] -> CoreM [CoreToDo]
install _ _ = return []
```

is certainly valid, but also certainly not what anyone really wants.

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 GhcPlugins

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
```

Using Annotations

Previously we discussed annotation pragmas ([Source annotations](#) (page 473)), 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 GhcPlugins
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)

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 anns [] (varUnique bndr)

```

Please see the GHC API documentation for more about how to use internal APIs, etc.

11.3.4 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 364)).

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
  , tcPluginStop  :: s -> TcPluginM ()
  }

type TcPluginSolver = [Ct] -> [Ct] -> [Ct] -> TcPluginM TcPluginResult

data TcPluginResult = TcPluginContradiction [Ct] | TcPluginOk [(EvTerm,Ct)] [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 two fields.
- During constraint solving, GHC repeatedly calls tcPluginSolve. This function is provided with the current set of constraints, and should return a TcPluginResult 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.

- 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.

Constraint solving with plugins

The key component of a typechecker plugin is a function of type `TcPluginSolver`, like this:

```
solve :: [Ct] -> [Ct] -> [Ct] -> TcPluginM TcPluginResult
solve givens deriveds wanteds = ...
```

This function will be invoked at two points in the constraint solving process: after simplification of given constraints, and after unflattening of wanted constraints. The two phases can be distinguished because the deriveds and wanteds will be empty in the first case. In each case, the plugin should either

- return `TcPluginContradiction` with a list of impossible constraints (which must be a subset of those passed in), so they can be turned into errors; or
- return `TcPluginOk` with lists of solved and new constraints (the former must be a subset of those passed in and must be supplied with corresponding evidence terms).

If the plugin cannot make any progress, it should return `TcPluginOk [] []`. 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 and derived 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.

11.3.5 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 66) 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 `GhcPlugins.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.

```
module DoNothing.FrontendPlugin (frontendPlugin) where
import GhcPlugins

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.

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 6) 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 501), which describes Glasgow Haskell’s shortcomings vs. the Haskell language definition, may also be of interest.)

12.1 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. If you turn on `-XNoImplicitPrelude-XNoImplicitPrelude` option, the compiler will almost surely die, unless you know what you are doing.

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.

12.2 When your program “does the wrong thing”

(For advice about overly slow or memory-hungry Haskell programs, please see [Advice on: sooner, faster, smaller, thriftier](#) (page 227)).

“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 161).

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 6)).

“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).

DEBUGGING COMPILED PROGRAMS

Since the 7.10 release GHC can emit a debugging information to help debugging tools understand the code that GHC produces. This debugging information is useable by most UNIX debugging tools.

-g
-g{n}

Since 7.10, numeric levels since 8.0

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, with 0 disabling debug information production. Levels 1 through 3 are functionally equivalent.

13.1 Tutorial

Let's consider a simple example,

```
1  -- fib.hs
2  fib :: Int -> Int
3  fib 0 = 0
4  fib 1 = 1
5  fib n = fib (n-1) + fib (n-2)
6
7  main :: IO ()
8  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 integerzm_gmp_GHCziIntegerziType_eqInteger_info () at libraries/integer-gmp
#4  0x00000000006406eca in roz_info () at Fib.hs:2
#5  0x00000000006eb0c0 in ?? ()
#6  0x000000000064f075 in cfri_info () at libraries/integer-gmp/src/GHC/Integer/Type.hs:412
#7  0x00000000006eb0c0 in ?? ()
#8  0x000000000064f075 in cfri_info () at libraries/integer-gmp/src/GHC/Integer/Type.hs:412
#9  0x00000000006eb0c0 in ?? ()
#10 0x000000000064eefe in integerzm_gmp_GHCziIntegerziType_plusInteger_info () at libraries/integer-g
...
#64 0x0000000000643ac8 in integerzm_gmp_GHCziIntegerziType_ltIntegerzh_info () at libraries/integer-g
#65 0x000000000064effcc in base_GHCziShow_zdwintegerToString_info () at libraries/base/GHC/Show.hs:44
#66 0x000000000064f0795 in base_GHCziShow_zdfShowIntegerzuzdcshow_info () at libraries/base/GHC/Show.
#67 0x000000000064892b in cdGW_info () at libraries/base/GHC/IO/Handle/Text.hs:595
#68 0x00000000006419cb2 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 targetting 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 integerzm_gmp_GHCziIntegerziType_ltIntegerzh_info () at libraries/integer-g
#3  0x00000000004effcc in base_GHCziShow_zdwintegerToString_info () at libraries/base/GHC/Show.hs:44
#4  0x00000000004f0795 in base_GHCziShow_zdfShowIntegerzdcshow_info () at libraries/base/GHC/Show.
#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.

13.2 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.

13.3 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)
0x6e6126 backtrace_handler (rts/posix/Signals.c:541.0)
0x7f317677017f (null) (/lib/x86_64-linux-gnu/libc-2.19.so)
0x642e1c integerzmghp_GHCziIntegerziType_eqIntegerzh_info (libraries/integer-gmp/src/GHC/In
0x643023 integerzmghp_GHCziIntegerziType_eqInteger_info (libraries/integer-gmp/src/GHC/In
0x406eca roz_info (/opt/exp/ghc/ghc-dwarf/Fib.hs:2.1)
0x6eafc0 stg_upd_frame_info (rts/Updates.cmm:31.1)
0x64ee06 integerzmghp_GHCziIntegerziType_plusInteger_info (libraries/integer-gmp/src/GHC/In
0x6eafc0 stg_upd_frame_info (rts/Updates.cmm:31.1)
...
0x6439d0 integerzmghp_GHCziIntegerziType_ltIntegerzh_info (libraries/integer-gmp/src/GHC/In
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)
```

13.4 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.

13.4.1 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.

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

13.5 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](#)*.

OTHER HASKELL UTILITY PROGRAMS

This section describes other program(s) which we distribute, that help with the Great Haskell Programming Task.

14.1 “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.

14.2 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.

14.2.1 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 494)).
- cross-safe** Restrict the `.hsc` directives to those supported by the `--cross-compile` mode (see [Cross-compilation](#) (page 494)). 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:

<code>.hs</code>	Haskell file
<code>_hsc.h</code>	C header
<code>_hsc.c</code>	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.

14.2.2 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 (message) 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.

14.2.3 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.

14.2.4 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

RUNNING GHC ON WIN32 SYSTEMS

15.1 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.

15.2 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.

15.3 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 <code>Debug.Trace.trace</code> is alright because it uses Windows debugging output support rather than <code>stderr</code> .
--

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!

15.4 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.

15.5 Using GHC (and other GHC-compiled executables) with Cygwin

15.5.1 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`).

15.5.2 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.

15.5.3 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 `R00T=$(pwd)` in make-file 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
```

15.6 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 192).

- 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.

15.6.1 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.

15.6.2 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 467); 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:

```
-- 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:

```
// 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);
}

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 [Trac #3605](#)).

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.

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
```

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 483)) 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

By default, GHC mainly aims to behave (mostly) like a Haskell 2010 compiler, although you can tell it to try to behave like a particular version of the language with the `-XHaskell98` and `-XHaskell2010` flags. The known deviations from the standards are described below. Unless otherwise stated, the deviation applies in Haskell 98, Haskell 2010 and the default modes.

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 `.\`.

Context-free syntax

- In Haskell 98 mode and by default (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` extension.

- 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)
```

- The Haskell Report allows you to put a unary `-` preceding certain expressions headed by keywords, allowing constructs like `- case x of ...` or `- do { ... }`. GHC does not allow this. Instead, unary `-` is allowed before only expressions that could potentially be applied as a function.

Expressions and patterns

In its default mode, 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
```

This improves efficiency slightly but significantly for most programs, and is bad for only a few. To suppress this bogus “optimisation” use `-fpedantic-bottoms`.

Declarations and bindings

In its default mode, GHC does not accept datatype contexts, as it has been decided to remove them from the next version of the language standard. This behaviour can be controlled with the `DatatypeContexts` extension. See [Data type contexts](#) (page 264).

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
```

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 152). 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.”

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 *-XRankNTypes* (page 380) 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.

In Prelude support

Arbitrary-sized tuples Tuples are currently limited to size 100. However, standard instances for tuples (Eq, Ord, Bounded, Ix, Read, and Show) are available *only* up to 16-tuples.

This limitation is easily subvertible, so please ask if you get stuck on it.

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`.

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 [Trac #13693](#).

Operator sections

The Haskell Report demands that, for infix operators `%`, the following identities hold:

```
:: (% expr) = x -> x % expr (expr %) = x -> expr % x
```

However, the second law is violated in the presence of undefined operators,

```
:: (%) = error "urk" (() %) seq () - urk (x -> () % x) seq () - OK, result ()
```

The operator section is treated like function application of an undefined function, while the lambda form is in WHNF that contains an application of an undefined function.

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, so all operations on `Int` happen modulo $2^{(n)}$ 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 $(\text{abs } x)$, 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)

16.2 Known bugs or infelicities

The bug tracker lists bugs that have been reported in GHC but not yet fixed: see the [GHC Trac](#). 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 [Trac #367](#) for further discussion.

If you are hit by this, you may want to compile the affected module with `-fno-omit-yields` (page 93) (see `-f*`: [platform-independent flags](#) (page 88)). 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` 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 90) 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 74) option (see *Platform-specific Flags* (page 74)), 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 optimization can result in non-obvious changes in evaluation ordering which may hide exceptions, even with *-fpedantic-bottoms* (page 93) (see, e.g., [Trac #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 76) 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 [Trac #11197](#).

- Despite appearances `*` and `Constraint` aren't really distinct kinds in the compiler's internal representation and can be unified producing unexpected results. See [Trac #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 [Trac #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 (# relocations found: 3070)
```

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.

EVENTLOG ENCODINGS

This section documents the encodings of the events emitted to GHC's *event log* (page 143). 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.

17.1 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.

17.1.1 Metadata event types

Beginning of sample stream

A single fixed-width event emitted during program start-up describing the samples that follow.

- EVENT_HEAP_PROF_BEGIN * Word8: Profile ID * Word64: Sampling period in nanoseconds * Word32: Sample break-down type. One of,
 - HEAP_PROF_BREAKDOWN_COST_CENTER (output from *-hc* (page 212))
 - HEAP_PROF_BREAKDOWN_CLOSURE_DESCR (output from *-hd* (page 212))
 - HEAP_PROF_BREAKDOWN_RETAINER (output from *-hr* (page 213))
 - HEAP_PROF_BREAKDOWN_MODULE (output from *-hm* (page 212))
 - HEAP_PROF_BREAKDOWN_TYPE_DESCR (output from *-hy* (page 213))
 - HEAP_PROF_BREAKDOWN_BIOGRAPHY (output from *-hb* (page 213))
 - HEAP_PROF_BREAKDOWN_CLOSURE_TYPE (output from *-hT* (page 143))
 - String: Module filter
 - String: Closure description filter
 - String: Type description filter
 - String: Cost centre filter
 - String: Cost centre stack filter

- String: Retainer filter
- String: Biography filter

Cost centre definitions

A variable-length packet produced once for each cost centre,

- `EVENT_HEAP_PROF_COST_CENTRE` * Word32: cost centre number * String: label * String: module * String: source location * Word8: flags
 - bit 0: is the cost-centre a CAF?

17.1.2 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 mark the beginning of a new sample with an `EVENT_HEAP_PROF_SAMPLE_BEGIN` event,

- `EVENT_HEAP_PROF_SAMPLE_BEGIN` * Word64: sample number

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.

Cost-centre break-down

A variable-length packet encoding a heap profile sample broken down by,

- cost-centre (-hc)
- `EVENT_HEAP_PROF_SAMPLE_COST_CENTRE` * Word8: Profile ID * Word64: heap residency in bytes * Word8: stack depth * Word32[]: cost centre stack starting with inner-most (cost centre numbers)

String break-down

A variable-length event encoding a heap sample broken down by,

- type description (-hy)
- closure description (-hd)
- module (-hm)
- `EVENT_HEAP_PROF_SAMPLE_STRING` * Word8: Profile ID * Word64: heap residency in bytes * String: type or closure description, or module name

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 ??).

18.1 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,

Cheloni
The taxonomic order consisting of modern turtles

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.

18.1.1 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

^^^^^

## 18.1.2 Formatting code

### 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)
```

### 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!"
```

## 18.1.3 Links

### 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.

### To GHC Trac resources

There are special macros for conveniently linking to GHC Trac Wiki articles and tickets,

See `:ghc-wiki:Commentary/Latedmd` for details on demand analysis.  
See the `:ghc-wiki:coding style <Commentary/CodingStyle>` for guidelines.  
See the `:ghc-ticket:123` for further discussion.  
See the `:ghc-ticket:this bug <123>` for what happens when this fails.

### To external resources

External links can be written in either of these ways,

See the ``GHC Wiki <http://ghc.haskell.org/wiki>`_` for details.  
See the ``GHC Wiki`_` for details.  
.. `_GHC Wiki: http://ghc.haskell.org/wiki`

### 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 <Control-Applicative.html>` for details.

### 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,  $J = -D \frac{d\varphi}{dx}$ , ...

### 18.1.4 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.

## 18.2 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)
```

## 18.3 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

## 18.4 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`),

### 18.4.1 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}

    :default: 20 milliseconds

    Sets the context switch interval to {s} seconds.
```

Will be rendered as,

```
-C {seconds}

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 101) 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}
```

### 18.4.2 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.

## 18.5 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).

## 18.6 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](#)



## INDICES AND TABLES

- `genindex`
- `search`



## BIBLIOGRAPHY

- [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.
- [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] “[Type 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.
- [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.





- +RTS, 131
- +t option
  - in GHCi, 24
- RTS, 131
- exclude-module={file}
  - GHC option, 162
- force
  - ghc-pkg option, 175
- frontend {module}
  - GHC option, 66
- generate-crash-dumps
  - RTS option, 133
- generate-stack-traces=<yes|no>
  - RTS option, 133
- global
  - ghc-pkg option, 175
- help
  - GHC option, 66
  - ghc-pkg option, 175
- info
  - GHC option, 67
  - RTS option, 146
- install-seh-handlers={yes|no}
  - RTS option, 133
- install-signal-handlers={yes|no}
  - RTS option, 133
- interactive, 42
  - GHC option, 66
- ipid
  - ghc-pkg option, 175
- long-gc-sync
  - RTS option, 139
- long-gc-sync=<seconds>
  - RTS option, 139
- machine-readable
  - RTS option, 140
- make
  - GHC option, 66
  - mode of GHC, 67
- mk-dll
  - GHC option, 66
- numa
  - RTS option, 139
- numa=<mask>
  - RTS option, 139
- numeric-version
  - GHC option, 67
- package-key
  - ghc-pkg option, 175
- print-libdir
  - GHC option, 67
- show-iface {file}
  - GHC option, 66, 151
- show-options
  - GHC option, 67
- supported-extensions
  - GHC option, 67
- supported-languages
  - GHC option, 67
- user
  - ghc-pkg option, 175
- verbose
  - ghc-pkg option, 175
- version
  - GHC option, 67
  - ghc-pkg option, 175
- ?
  - GHC option, 66
  - ghc-pkg option, 175
  - hp2ps command line option, 217
- A {size}
  - RTS option, 134
- AL {size}
  - RTS option, 134
- A{size}
  - RTS option, 227
- B
  - RTS option, 144
- C, 69
  - GHC option, 66
- C {s}
  - RTS option, 99
- D {x}
  - RTS option, 144

- D(symbol)[=<value>]
  - GHC option, [183](#)
- E, [69](#)
  - GHC option, [66](#)
- F
  - GHC option, [185](#)
- F <factor>
  - RTS option, [135](#)
- G RTS option, [231](#)
- G <generations>
  - RTS option, [136](#)
- H
  - RTS option, [227](#)
- H [<size>]
  - RTS option, [137](#)
- H <size>
  - GHC option, [73](#)
- I <seconds>
  - RTS option, [137](#)
- I<dir>
  - GHC option, [183](#)
- K <size>
  - RTS option, [138](#)
- L <dir>
  - GHC option, [188](#)
- L <num>
  - RTS option, [214](#)
- L <n>
  - RTS option, [143](#)
- M
  - GHC option, [66](#)
- M <size>
  - RTS option, [138](#)
- M<size>
  - RTS option, [139](#)
- N <x>
  - RTS option, [100](#)
- O
  - GHC option, [88](#)
- O <size>
  - RTS option, [135](#)
- O0
  - GHC option, [88](#)
- O1
  - GHC option, [88](#)
- O2
  - GHC option, [88](#)
- Odph
  - GHC option, [88](#)
- P
  - RTS option, [208](#)
- R <size>
  - RTS option, [214](#)
- RTS, [131](#)
- Rghc-timing
  - GHC option, [73](#)
- S, [69](#)
  - GHC option, [66](#)
- S RTS option, [231](#)
- S [<file>]
  - RTS option, [140](#)
- T
  - RTS option, [140](#)
- U(symbol)
  - GHC option, [183](#)
- V
  - GHC option, [67](#)
  - ghc-pkg option, [175](#)
- V <secs>
  - RTS option, [208](#)
- W
  - GHC option, [75](#)
- Wall
  - GHC option, [75](#)
- Wall-missed-specialisations
  - GHC option, [77](#)
- Wamp
  - GHC option, [78](#)
- Wcompat
  - GHC option, [75](#)
- Wcpp-undef
  - GHC option, [87](#)
- Wdeferred-out-of-scope-variables
  - GHC option, [77](#)
- Wdeferred-type-errors
  - GHC option, [76](#)
- Wdeprecated-flags
  - GHC option, [79](#)
- Wdeprecations
  - GHC option, [78](#)
- Wdodgy-exports
  - GHC option, [79](#)
- Wdodgy-foreign-imports
  - GHC option, [79](#)
- Wdodgy-imports
  - GHC option, [79](#)
- Wduplicate-constraints
  - GHC option, [80](#)
- Wduplicate-exports
  - GHC option, [80](#)
- Wempty-enumerations
  - GHC option, [79](#)
- Werror
  - GHC option, [75](#)
- Weverything
  - GHC option, [75](#)
- Whi-shadowing
  - GHC option, [80](#)

- Widentities
  - GHC option, [80](#)
- Wimplicit-prelude
  - GHC option, [81](#)
- Wincomplete-patterns
  - GHC option, [81](#)
- Wincomplete-record-updates
  - GHC option, [81](#)
- Wincomplete-uni-patterns
  - GHC option, [81](#)
- Winline-rule-shadowing
  - GHC option, [87](#)
- Wmissed-specialisations
  - GHC option, [77](#)
- Wmissing-export-lists
  - GHC option, [82](#)
- Wmissing-exported-signatures
  - GHC option, [83](#)
- Wmissing-exported-sigs
  - GHC option, [83](#)
- Wmissing-fields
  - GHC option, [82](#)
- Wmissing-home-modules
  - GHC option, [87](#)
- Wmissing-import-lists
  - GHC option, [82](#)
- Wmissing-local-signatures
  - GHC option, [83](#)
- Wmissing-local-sigs
  - GHC option, [83](#)
- Wmissing-methods
  - GHC option, [82](#)
- Wmissing-monadfail-instances
  - GHC option, [78](#)
- Wmissing-pattern-synonym-signatures
  - GHC option, [83](#)
- Wmissing-signatures
  - GHC option, [82](#)
- Wmonomorphism-restriction
  - GHC option, [84](#)
- Wname-shadowing
  - GHC option, [83](#)
- Wno-compat
  - GHC option, [75](#)
- Wnoncanonical-monad-instances
  - GHC option, [78](#)
- Wnoncanonical-monadfail-instances
  - GHC option, [78](#)
- Wnoncanonical-monoid-instances
  - GHC option, [78](#)
- Worphans
  - GHC option, [83](#)
- Woverflowed-literals
  - GHC option, [79](#)
- Woverlapping-patterns
  - GHC option, [83](#)
- Wpartial-fields
  - GHC option, [87](#)
- Wpartial-type-signatures
  - GHC option, [77](#)
- Wredundant-constraints
  - GHC option, [80](#)
- Wsafe
  - GHC option, [459](#)
- Wsemigroup
  - GHC option, [79](#)
- Wsimplifiable-class-constraints
  - GHC option, [84](#)
- Wtabs
  - GHC option, [84](#)
- Wtrustworthy-safe
  - GHC option, [459](#)
- Wtype-defaults
  - GHC option, [84](#)
- Wtyped-holes
  - GHC option, [76](#)
- Wunbanged-strict-patterns
  - GHC option, [87](#)
- Wunrecognised-pragmas
  - GHC option, [77](#)
- Wunrecognised-warning-flags
  - GHC option, [76](#)
- Wunsafe
  - GHC option, [459](#)
- Wunsupported-calling-conventions
  - GHC option, [79](#)
- Wunsupported-llvm-version
  - GHC option, [84](#)
- Wunticked-promoted-constructors
  - GHC option, [84](#)
- Wunused-binds
  - GHC option, [84](#)
- Wunused-do-bind
  - GHC option, [86](#)
- Wunused-foralls
  - GHC option, [86](#)
- Wunused-imports
  - GHC option, [85](#)
- Wunused-local-binds
  - GHC option, [85](#)
- Wunused-matches
  - GHC option, [85](#)
- Wunused-pattern-binds
  - GHC option, [85](#)
- Wunused-top-binds
  - GHC option, [85](#)
- Wunused-type-patterns
  - GHC option, [86](#)

- Wwarn
  - GHC option, [76](#)
- Wwarnings-deprecations
  - GHC option, [77](#)
- Wwrong-do-bind
  - GHC option, [86](#)
- XEmptyDataDeriving
  - GHC option, [286](#)
- XHexFloatLiterals
  - GHC option, [242](#)
- XUndecidableInstances, [323](#)
- Z
  - RTS option, [145](#)
- b
  - hp2ps command line option, [216](#)
- c, [69](#)
  - GHC option, [66](#), [187](#)
  - hp2ps command line option, [217](#)
  - RTS option, [135](#)
- c (n)
  - RTS option, [135](#)
- clear-package-db
  - GHC option, [169](#)
- cpp
  - GHC option, [183](#)
- cpp vs string gaps, [185](#)
- d
  - hp2ps command line option, [216](#)
- dcmm-lint
  - GHC option, [202](#)
- dcore-lint
  - GHC option, [202](#)
- ddump-asm
  - GHC option, [200](#)
- ddump-asm-expanded
  - GHC option, [200](#)
- ddump-asm-liveness
  - GHC option, [200](#)
- ddump-asm-native
  - GHC option, [200](#)
- ddump-asm-regalloc
  - GHC option, [200](#)
- ddump-asm-regalloc-stages
  - GHC option, [200](#)
- ddump-asm-stats
  - GHC option, [200](#)
- ddump-bcos
  - GHC option, [200](#)
- ddump-cmm
  - GHC option, [200](#)
- ddump-cmm-caf
  - GHC option, [199](#)
- ddump-cmm-cbe
  - GHC option, [199](#)
- ddump-cmm-cfg
  - GHC option, [199](#)
- ddump-cmm-cps
  - GHC option, [200](#)
- ddump-cmm-from-stg
  - GHC option, [199](#)
- ddump-cmm-info
  - GHC option, [200](#)
- ddump-cmm-proc
  - GHC option, [199](#)
- ddump-cmm-procmap
  - GHC option, [199](#)
- ddump-cmm-raw
  - GHC option, [199](#)
- ddump-cmm-sink
  - GHC option, [199](#)
- ddump-cmm-sp
  - GHC option, [199](#)
- ddump-cmm-split
  - GHC option, [199](#)
- ddump-cmm-switch
  - GHC option, [199](#)
- ddump-cmm-verbose
  - GHC option, [199](#)
- ddump-core-stats
  - GHC option, [198](#)
- ddump-cse
  - GHC option, [198](#)
- ddump-deriv
  - GHC option, [197](#)
- ddump-ds
  - GHC option, [198](#)
- ddump-ec-trace
  - GHC option, [197](#)
- ddump-foreign
  - GHC option, [200](#)
- ddump-hi
  - GHC option, [151](#)
- ddump-hi-diffs, [484](#)
  - GHC option, [151](#)
- ddump-if-trace
  - GHC option, [197](#)
- ddump-inlinings
  - GHC option, [198](#)
- ddump-json
  - GHC option, [196](#)
- ddump-llvm
  - GHC option, [200](#)
- ddump-minimal-imports
  - GHC option, [151](#)
- ddump-mod-cycles
  - GHC option, [162](#)
- ddump-occur-anal
  - GHC option, [199](#)

- ddump-opt-cmm
  - GHC option, [200](#)
- ddump-parsed
  - GHC option, [197](#)
- ddump-parsed-ast
  - GHC option, [197](#)
- ddump-prep
  - GHC option, [199](#)
- ddump-rn
  - GHC option, [197](#)
- ddump-rn-ast
  - GHC option, [197](#)
- ddump-rn-stats
  - GHC option, [197](#)
- ddump-rn-trace
  - GHC option, [197](#)
- ddump-rule-firings
  - GHC option, [198](#)
- ddump-rule-rewrites
  - GHC option, [198](#)
- ddump-rules
  - GHC option, [198](#)
- ddump-simpl
  - GHC option, [198](#)
- ddump-simpl-iterations
  - GHC option, [198](#)
- ddump-simpl-stats
  - GHC option, [198](#)
- ddump-spec
  - GHC option, [198](#)
- ddump-splices
  - GHC option, [197](#)
- ddump-stg
  - GHC option, [199](#)
- ddump-str-signatures
  - GHC option, [198](#)
- ddump-stranal
  - GHC option, [198](#)
- ddump-tc
  - GHC option, [197](#)
- ddump-tc-ast
  - GHC option, [197](#)
- ddump-tc-trace
  - GHC option, [197](#)
- ddump-timings
  - GHC option, [196](#)
- ddump-to-file
  - GHC option, [196](#)
- ddump-types
  - GHC option, [197](#)
- ddump-vect
  - GHC option, [198](#)
- ddump-vt-trace
  - GHC option, [199](#)
- ddump-worker-wrapper
  - GHC option, [198](#)
- debug
  - GHC option, [189](#)
- dep-makefile (file)
  - GHC option, [162](#)
- dep-suffix (suffix)
  - GHC option, [162](#)
- dfaststring-stats
  - GHC option, [196](#)
- dinitial-unique=(s)
  - GHC option, [202](#)
- distrust (pkg)
  - GHC option, [457](#)
- distrust-all-packages
  - GHC option, [457](#)
- dno-debug-output
  - GHC option, [201](#)
- dppr-case-as-let
  - GHC option, [201](#)
- dppr-cols=(n)
  - GHC option, [201](#)
- dppr-debug
  - GHC option, [196](#)
- dppr-user-length
  - GHC option, [201](#)
- dshow-passes
  - GHC option, [196](#)
- dstg-lint
  - GHC option, [202](#)
- dsuppress-all
  - GHC option, [201](#)
- dsuppress-coercions
  - GHC option, [201](#)
- dsuppress-idinfo
  - GHC option, [201](#)
- dsuppress-module-prefixes
  - GHC option, [201](#)
- dsuppress-stg-free-vars
  - GHC option, [202](#)
- dsuppress-ticks
  - GHC option, [201](#)
- dsuppress-type-applications
  - GHC option, [201](#)
- dsuppress-type-signatures
  - GHC option, [201](#)
- dsuppress-unfoldings
  - GHC option, [201](#)
- dsuppress-uniques
  - GHC option, [201](#)
- dsuppress-var-kinds
  - GHC option, [201](#)
- dth-dec-file=(file)
  - GHC option, [197](#)

- dumpdir <dir>  
GHC option, 150
- dunique-increment=<i>  
GHC option, 202
- dverbose-core2core  
GHC option, 198
- dverbose-stg2stg  
GHC option, 199
- dylib-install-name <path>  
GHC option, 191
- dynamic  
GHC option, 188
- dynamic-too  
GHC option, 187
- dynload  
GHC option, 189
- e <expr>  
GHC option, 66
- eventlog  
GHC option, 190
- e(float)[in|mm|pt]  
hp2ps command line option, 216
- f  
ghc-pkg option, 175
- fPIC  
GHC option, 186
- fPIE  
GHC option, 186
- f\\* options (GHC), 88
- falignment-sanitisation  
GHC option, 202
- fasm  
GHC option, 186
- fbreak-on-error  
GHC option, 41
- fbreak-on-exception  
GHC option, 40
- fbyte-code  
GHC option, 186
- fcall-arity  
GHC option, 89
- fcase-folding  
GHC option, 89
- fcase-merge  
GHC option, 88
- fcatch-bottoms  
GHC option, 202
- fcmm-elim-common-blocks  
GHC option, 89
- fcmm-sink  
GHC option, 89
- fcpr-anal  
GHC option, 89
- fcross-module-specialise  
GHC option, 96
- fcse  
GHC option, 89
- fdefer-out-of-scope-variables  
GHC option, 77
- fdefer-type-errors  
GHC option, 76
- fdefer-typed-holes  
GHC option, 76
- fdiagnostics-color=(always|auto|never)  
GHC option, 72
- fdiagnostics-show-caret  
GHC option, 73
- fdicts-cheap  
GHC option, 90
- fdicts-strict  
GHC option, 90
- fdmd-tx-dict-sel  
GHC option, 90
- fdo-eta-reduction  
GHC option, 90
- fdo-lambda-eta-expansion  
GHC option, 90
- feager-blackholing  
GHC option, 90
- fenable-rewrite-rules  
GHC option, 432
- ferror-spans  
GHC option, 73
- fexcess-precision  
GHC option, 90
- fexitification  
GHC option, 89
- fexpose-all-unfoldings  
GHC option, 90
- fexternal-interpreter  
GHC option, 58
- ffloat-in  
GHC option, 91
- fforce-recomp  
GHC option, 152
- ffull-laziness  
GHC option, 91
- ffun-to-thunk  
GHC option, 91
- fghci-hist-size=<n>  
GHC option, 40
- fglasgow-exts  
GHC option, 235
- fhelpful-errors  
GHC option, 77
- fhide-source-paths  
GHC option, 70
- fhpc

- GHC option, 221
- fignore-asserts
  - GHC option, 91
- fignore-hpc-changes
  - GHC option, 152
- fignore-interface-pragmas
  - GHC option, 91
- fignore-optim-changes
  - GHC option, 152
- flate-dmd-anal
  - GHC option, 91
- fliberate-case
  - GHC option, 92
- fliberate-case-threshold=*(n)*
  - GHC option, 92
- flvm
  - GHC option, 186
- flvm-fill-undef-with-garbage
  - GHC option, 202
- flvm-pass-vectors-in-regs
  - GHC option, 92
- flocal-ghci-history
  - GHC option, 42
- floopification
  - GHC option, 92
- fmax-inline-alloc-size=*(n)*
  - GHC option, 92
- fmax-inline-memcpy-insns=*(n)*
  - GHC option, 92
- fmax-inline-memset-insns=*(n)*
  - GHC option, 92
- fmax-pmcheck-iterations=*(n)*
  - GHC option, 81
- fmax-relevant-binds=*(n)*
  - GHC option, 92
- fmax-simplifier-iterations=*(n)*
  - GHC option, 93
- fmax-uncovered-patterns=*(n)*
  - GHC option, 93
- fmax-valid-substitutions=*(n)*
  - GHC option, 92
- fmax-worker-args=*(n)*
  - GHC option, 93
- fno-\* options (GHC), 88
- fno-code
  - GHC option, 186
- fno-embed-manifest
  - GHC option, 191
- fno-gen-manifest
  - GHC option, 191
- fno-implicit-import-qualified, 28
- fno-opt-coercion
  - GHC option, 93
- fno-pre-inlining
  - GHC option, 93
- fno-prof-auto
  - GHC option, 208
- fno-prof-cafs
  - GHC option, 208
- fno-prof-count-entries
  - GHC option, 208
- fno-shared-implib
  - GHC option, 191
- fno-state-hack
  - GHC option, 93
- fobject-code
  - GHC option, 186
- fomit-interface-pragmas
  - GHC option, 93
- fomit-yields
  - GHC option, 93
- foptimal-applicative-do
  - GHC option, 250
- fpackage-trust, 456
  - GHC option, 459
- fpedantic-bottoms
  - GHC option, 93
- fplugin-opt=*(module):{args}*
  - GHC option, 475
- fplugin=*(module)*
  - GHC option, 475
- fprint-bind-result
  - GHC option, 22
- fprint-equality-relations
  - GHC option, 71
- fprint-expanded-synonyms
  - GHC option, 71
- fprint-explicit-coercions
  - GHC option, 71
- fprint-explicit-foralls
  - GHC option, 70
- fprint-explicit-kinds
  - GHC option, 71
- fprint-explicit-runtime-rep
  - GHC option, 362
- fprint-explicit-runtime-reps
  - GHC option, 71
- fprint-potential-instances
  - GHC option, 70
- fprint-typechecker-elaboration
  - GHC option, 72
- fprint-unicode-syntax
  - GHC option, 70
- fprof-auto
  - GHC option, 207
- fprof-auto-calls
  - GHC option, 207
- fprof-auto-exported

- GHC option, [207](#)
- fprof-auto-top
  - GHC option, [207](#)
- fprof-cafs, [207](#)
  - GHC option, [207](#)
- framework <name>
  - GHC option, [187](#)
- framework-path <dir>
  - GHC option, [188](#)
- fregs-graph
  - GHC option, [94](#)
- fregs-iterative
  - GHC option, [94](#)
- fshow-hole-constraints
  - GHC option, [387](#)
- fshow-loaded-modules
  - GHC option, [18](#)
- fshow-warning-groups
  - GHC option, [76](#)
- fsimpl-tick-factor=<n>
  - GHC option, [94](#)
- fsimplifier-phases=<n>
  - GHC option, [94](#)
- fsolve-constant-dicts
  - GHC option, [96](#)
- fspec-constr
  - GHC option, [94](#)
- fspec-constr-count=<n>
  - GHC option, [95](#)
- fspec-constr-keen
  - GHC option, [95](#)
- fspec-constr-threshold=<n>
  - GHC option, [95](#)
- fspecialise
  - GHC option, [95](#)
- fspecialise-aggressively
  - GHC option, [96](#)
- fstatic-argument-transformation
  - GHC option, [96](#)
- fstg-cse
  - GHC option, [90](#)
- fstrictness
  - GHC option, [96](#)
- fstrictness-before=<n>
  - GHC option, [97](#)
- funbox-small-strict-fields
  - GHC option, [97](#)
- funbox-strict-fields
  - GHC option, [97](#)
- funfolding-creation-threshold=<n>
  - GHC option, [97](#)
- funfolding-dict-discount=<n>
  - GHC option, [98](#)
- funfolding-fun-discount=<n>
  - GHC option, [98](#)
- funfolding-keenness-factor=<n>
  - GHC option, [98](#)
- funfolding-use-threshold0 option, [231](#)
- funfolding-use-threshold=<n>
  - GHC option, [98](#)
- fvectorisation-avoidance, [98](#)
  - GHC option, [98](#)
- fvectorise
  - GHC option, [98](#)
- fvia-C, [181](#)
- fwhole-archive-hs-libs
  - GHC option, [192](#)
- fwrite-interface
  - GHC option, [186](#)
- g
  - GHC option, [485](#)
  - hp2ps command line option, [217](#)
- ghci-script
  - GHC option, [57](#)
- ghcversion-file <path to ghcversion.h>
  - GHC option, [74](#)
- global-package-db
  - GHC option, [169](#)
- g<n>
  - GHC option, [485](#)
- h
  - RTS option, [143](#), [212](#)
- hT
  - RTS option, [143](#), [212](#)
- hb
  - RTS option, [213](#)
- hc
  - RTS option, [212](#)
- hcsuf <suffix>
  - GHC option, [150](#)
- hd
  - RTS option, [212](#)
- hide-all-packages
  - GHC option, [166](#)
- hide-all-plugin-packages
  - GHC option, [476](#)
- hide-package <pkg>
  - GHC option, [166](#)
- hidir <dir>
  - GHC option, [150](#)
- hisuf <suffix>
  - GHC option, [150](#)
- hm
  - RTS option, [212](#)
- hr
  - RTS option, [213](#)
- hy
  - RTS option, [213](#)



- h(break-down), 216
- i
  - GHC option, 148
- i {secs}
  - RTS option, 214
- ignore-dot-ghci
  - GHC option, 57
- ignore-package {pkg}
  - GHC option, 166
- include-pkg-deps
  - GHC option, 162
- interactive-print {expr}
  - GHC option, 32
- i{dir}[:{dir}]\*
  - GHC option, 148
- j[{n}]
  - GHC option, 68
- kb {size}
  - RTS option, 138
- kc {size}
  - RTS option, 138
- keep-hc-file
  - GHC option, 150
- keep-hc-files
  - GHC option, 150
- keep-hi-files
  - GHC option, 150
- keep-llvm-file
  - GHC option, 150
- keep-llvm-files
  - GHC option, 150
- keep-o-files
  - GHC option, 151
- keep-s-file
  - GHC option, 151
- keep-s-files
  - GHC option, 151
- keep-tmp-files
  - GHC option, 151
- ki {size}
  - RTS option, 137
- l
  - hp2ps command line option, 217
  - RTS option, 213
- l {flags}
  - RTS option, 143
- l {lib}
  - GHC option, 187
- m {n}
  - RTS option, 138
- m\\* options, 74
- main-is {thing}
  - GHC option, 189
- maxN {x}
  - RTS option, 100
- msse2
  - GHC option, 74
- msse2 option, 507
- msse4.2
  - GHC option, 74
- m(int)
  - hp2ps command line option, 217
- n {size}
  - RTS option, 135
- no-auto-link-packages
  - GHC option, 166
- no-global-package-db
  - GHC option, 169
- no-hs-main
  - GHC option, 189
- no-rtsopts-suggestions
  - GHC option, 191
- no-user-package-db
  - GHC option, 169
- o {file}
  - GHC option, 149
- odir {dir}
  - GHC option, 149
- ohi {file}
  - GHC option, 149
- optF {option}
  - GHC option, 182
- optL {option}
  - GHC option, 182
- optP {option}
  - GHC option, 182
- opta {option}
  - GHC option, 182
- optc {option}
  - GHC option, 182
- optdll {option}
  - GHC option, 182
- opti {option}
  - GHC option, 183
- optl {option}
  - GHC option, 182
- optlc {option}
  - GHC option, 182
- optlo {option}
  - GHC option, 182
- optwindres {option}
  - GHC option, 183
- osuf {suffix}
  - GHC option, 150
- outputdir {dir}
  - GHC option, 150
- p
  - hp2ps command line option, 217

- RTS option, 203, 208
- pa
  - RTS option, 208
- package (name)
  - GHC option, 187
- package (pkg)
  - GHC option, 165
- package-db
  - ghc-pkg option, 175
- package-db (file)
  - GHC option, 169
- package-env (file)|(name)
  - GHC option, 170
- package-id (unit-id)
  - GHC option, 166
- pgmF (cmd)
  - GHC option, 182
- pgmL (cmd)
  - GHC option, 181
- pgmP (cmd)
  - GHC option, 181
- pgma (cmd)
  - GHC option, 182
- pgmc (cmd)
  - GHC option, 181
- pgmdll (cmd)
  - GHC option, 182
- pgmi (cmd)
  - GHC option, 182
- pgml (cmd)
  - GHC option, 182
- pgmlc (cmd)
  - GHC option, 181
- pgmlibtool (cmd)
  - GHC option, 182
- pgmlo (cmd)
  - GHC option, 181
- pgms (cmd)
  - GHC option, 182
- pgmwindres (cmd)
  - GHC option, 182
- pie
  - GHC option, 192
- pj
  - RTS option, 208
- plugin-package (pkg)
  - GHC option, 476
- plugin-package-id (pkg-id)
  - GHC option, 476
- po (stem)
  - RTS option, 208
- prof
  - GHC option, 207
- qa
  - RTS option, 100
- qb (gen)
  - RTS option, 136
- qq (gen)
  - RTS option, 136
- qm
  - RTS option, 101
- qn (x)
  - RTS option, 136
- r (file)
  - RTS option, 145
- rdynamic
  - GHC option, 192
- rtsopts[=(none|some|all)]
  - GHC option, 190
- s
  - hp2ps command line option, 217
- s [(file)]
  - RTS option, 140
- shared
  - GHC option, 188
- split-objs
  - GHC option, 188
- split-sections
  - GHC option, 188
- static
  - GHC option, 188
- staticlib
  - GHC option, 188
- stubdir (dir)
  - GHC option, 150
- t [(file)]
  - RTS option, 140
- this-unit-id (unit-id)
  - GHC option, 166
- threaded
  - GHC option, 189
- ticky
  - GHC option, 225
- tmpdir (dir)
  - GHC option, 151
- trust (pkg)
  - GHC option, 457
- t(float)
  - hp2ps command line option, 217
- user-package-db
  - GHC option, 169
- v
  - GHC option, 69, 227
  - ghc-pkg option, 175
- v [(flags)]
  - RTS option, 144
- v(n)
  - GHC option, 69

- w
  - GHC option, 75
- with-rtsopts={opts}
  - GHC option, 190
- x {suffix}
  - GHC option, 69
- xc
  - RTS option, 145, 209
- xm
  - RTS option, 134
- xm {address}
  - RTS option, 133
- xq {size}
  - RTS option, 134
- xt
  - RTS option, 214
- y
  - hp2ps command line option, 217
- .ghci
  - file, 56
- .haskeline
  - file, 57
- .hc files, saving, 150
- .hi files, 147
- .ll files, saving, 150
- .o files, 147
- .s files, saving, 150
- :
- {command}
  - GHCi command, 54
- GHCi command, 47
- :?
  - GHCi command, 47
- :abandon
  - GHCi command, 44
- :add
  - GHCi command, 44
- :all-types
  - GHCi command, 44
- :back
  - GHCi command, 44
- :break
  - GHCi command, 44
- :browse
  - GHCi command, 44
- :cd
  - GHCi command, 45
- :cmd
  - GHCi command, 45
- :complete
  - GHCi command, 45
- :continue
  - GHCi command, 46
- :ctags
  - GHCi command, 46
- :def
  - GHCi command, 46
- :delete
  - GHCi command, 47
- :edit
  - GHCi command, 47
- :etags
  - GHCi command, 47
- :force
  - GHCi command, 47
- :forward
  - GHCi command, 47
- :help
  - GHCi command, 47
- :history
  - GHCi command, 47
- :info
  - GHCi command, 47
- :issafe
  - GHCi command, 48
- :kind
  - GHCi command, 48
- :list
  - GHCi command, 48
- :list [{(module)}]
  - GHCi command, 48
- :load, 18
  - GHCi command, 48
- :loc-at
  - GHCi command, 49
- :main
  - GHCi command, 49
- :module
  - GHCi command, 49
- :print
  - GHCi command, 50
- :quit
  - GHCi command, 50
- :reload, 19
  - GHCi command, 50
- :run
  - GHCi command, 50
- :script
  - GHCi command, 50
- :set
  - command in GHCi, 54
  - GHCi command, 50
- :set +c
  - GHCi command, 54
- :set +m
  - GHCi command, 54
- :set +r
  - GHCi command, 54

- `:set +s`
  - GHCi command, 54
- `:set +t`
  - GHCi command, 54
- `:set args`
  - GHCi command, 50
- `:set editor`
  - GHCi command, 50
- `:set prog`
  - GHCi command, 50
- `:set prompt`
  - GHCi command, 50
- `:set prompt-cont`
  - GHCi command, 51
- `:set prompt-cont-function`
  - GHCi command, 51
- `:set prompt-function`
  - GHCi command, 51
- `:set stop`
  - GHCi command, 51
- `:seti, 54`
  - GHCi command, 51
- `:show`
  - GHCi command, 52
- `:show bindings`
  - GHCi command, 52
- `:show breaks`
  - GHCi command, 52
- `:show context`
  - GHCi command, 52
- `:show imports`
  - GHCi command, 52
- `:show language`
  - GHCi command, 52
- `:show modules`
  - GHCi command, 52
- `:show packages`
  - GHCi command, 52
- `:show paths`
  - GHCi command, 52
- `:showi language`
  - GHCi command, 52
- `:sprint`
  - GHCi command, 52
- `:step`
  - GHCi command, 52
- `:steplocal`
  - GHCi command, 52
- `:stepmodule`
  - GHCi command, 52
- `:trace`
  - GHCi command, 52
- `:type`
  - GHCi command, 53
- `:type +d`
  - GHCi command, 53
- `:type +v`
  - GHCi command, 53
- `:type-at`
  - GHCi command, 53
- `:undef`
  - GHCi command, 53
- `:unset`
  - GHCi command, 53
- `:uses`
  - GHCi command, 53
  - `__GLASGOW_HASKELL_LLVM__`, 184
  - `__GLASGOW_HASKELL_PATCHLEVEL1__`, 184
  - `__GLASGOW_HASKELL_PATCHLEVEL2__`, 184
  - `__GLASGOW_HASKELL_TH__`, 184
  - `__GLASGOW_HASKELL__`, 6, 183
  - `__PARALLEL_HASKELL__`, 185
  - “hs-boot” files, 153
- allocation area for large objects, size, 134
- allocation area, chunk size, 135
- allocation area, size, 134
- `AllowAmbiguousTypes`
  - Language Extension, 368
- AMP, 78
- ANN pragma, 473
  - on modules, 474
  - on types, 474
- apparently erroneous do binding, warning, 86
- Applicative do-notation, 248
- Applicative-Monad Proposal, 78
- `ApplicativeDo`
  - Language Extension, 248
- arguments
  - command-line, 64
- Arrows
  - Language Extension, 404
- ASCII, 147
- Assertions, 417
- assertions
  - disabling, 418
- author
  - package specification, 178
- auto
  - package specification, 178
- Bang patterns, 410
- `BangPatterns`
  - Language Extension, 411
- `BinaryLiterals`
  - Language Extension, 242

- binds, unused, [84](#), [85](#)
- bugs
  - reporting, [6](#)
- C calls, function headers, [467](#)
- C code generator, [181](#)
- C pre-processor options, [183](#)
- CAFs
  - in GHCi, [54](#)
- CApiFFI
  - Language Extension, [463](#)
- category
  - package specification, [178](#)
- cc-options
  - package specification, [179](#)
- Char
  - size of, [506](#)
- code coverage, [220](#)
- COLUMN
  - pragma, [426](#)
- command-line
  - arguments, [64](#)
  - order of arguments, [64](#)
- compacting garbage collection, [135](#)
- compilation phases, changing, [181](#)
- compiled code
  - in GHCi, [19](#)
- compiler problems, [483](#)
- compiling faster, [227](#)
- complete user-supplied kind signature, [354](#)
- concurrency, [447](#)
- Concurrent Haskell
  - using, [99](#)
- CONLIKE, [425](#)
- consistency checks, [202](#)
- Constant Applicative Form, [54](#)
- ConstrainedClassMethods
  - Language Extension, [313](#)
- ConstraintKinds
  - Language Extension, [366](#)
- constructor fields, strict, [97](#)
- copyright
  - package specification, [178](#)
- cost centres
  - automatically inserting, [207](#)
- cost-centre profiling, [203](#)
- CPP
  - Language Extension, [183](#)
- cpp, pre-processing with, [183](#)
- CUSK, [354](#)
- Custom printing function
  - in GHCi, [31](#)
- DataKinds
  - Language Extension, [349](#)
- DatatypeContexts
  - Language Extension, [264](#)
- debugger
  - in GHCi, [33](#)
- debugging options (for GHC), [195](#)
- default declarations, [31](#)
- defaulting mechanism, warning, [84](#)
- DefaultSignatures
  - Language Extension, [314](#)
- dependencies in Makefiles, [161](#)
- dependency-generation mode
  - of GHC, [66](#)
- depends
  - package specification, [179](#)
- DEPRECATED, [421](#)
- deprecated flags, [79](#)
- deprecations, [78](#)
  - warnings, [77](#)
- DeriveAnyClass
  - Language Extension, [302](#)
- DeriveDataTypeable
  - Language Extension, [295](#)
- DeriveFoldable
  - Language Extension, [292](#)
- DeriveFunctor
  - Language Extension, [289](#)
- DeriveGeneric
  - Language Extension, [440](#)
- DeriveLift
  - Language Extension, [296](#)
- DeriveTraversable
  - Language Extension, [294](#)
- DerivingStrategies
  - Language Extension, [304](#)
- description
  - package specification, [178](#)
- deterministic builds, [202](#)
- disabling
  - assertions, [418](#)
- DisambiguateRecordFields
  - Language Extension, [277](#)
- displaying type
  - in GHCi, [54](#)
- DLL-creation mode, [66](#)
- do binding, apparently erroneous, [86](#)
- do binding, unused, [86](#)
- do-notation
  - Applicative, [248](#)
  - in GHCi, [22](#)
- dumping GHC intermediates, [196](#)
- duplicate constraints, warning, [80](#)
- duplicate exports, warning, [80](#)
- DuplicateRecordFields

- Language Extension, 278
- dynamic
  - options, 55, 65
- Dynamic libraries
  - using, 192
- EDITOR, 47
- EmptyCase
  - Language Extension, 259
- EmptyDataDecls
  - Language Extension, 264
- encodings
  - of source files, 147
- environment file, 170
- environment variable
  - EDITOR, 47
  - for setting RTS options, 132
  - GHC\_PACKAGE\_PATH, 168, 169, 172, 173
  - GHCRTS, 131, 132, 190
  - HOME, 44, 45
  - HPCTIXFILE, 221, 225
  - LD\_LIBRARY\_PATH, 43, 195
  - PATH, 169, 186
  - RPATH, 195
  - RUNPATH, 195
  - TMPDIR, 151
- eval mode
  - of GHC, 66
- eventlog
  - and heap profiling, 213
- eventlog files, 143
- EventLogWriter (C type), 133
- EventLogWriter.flushEventLog (C member), 133
- EventLogWriter.initEventLogWriter (C member), 133
- EventLogWriter.stopEventLogWriter (C member), 133
- EventLogWriter.writeEventLog (C member), 133
- events, 143
- ExistentialQuantification
  - Language Extension, 267
- ExplicitForAll
  - Language Extension, 367
- ExplicitNamespaces
  - Language Extension, 262
- export lists, duplicates, 80
- export lists, missing, 82
- exposed
  - package specification, 178
- exposed-modules
  - package specification, 178
- extended list comprehensions, 252
- ExtendedDefaultRules
  - Language Extension, 30
- extensions
  - options controlling, 233
- extra-libraries
  - package specification, 179
- faster compiling, 227
- faster programs, how to produce, 228
- FFI
  - GHCi support, 17
- fields, missing, 82
- file names
  - of source files, 147
- file suffixes for GHC, 65
- filenames
  - of modules, 18
- finding interface files, 148
- FlexibleContexts
  - Language Extension, 312
- FlexibleInstances
  - Language Extension, 321
- Floating point
  - and the FFI, 472
- floating-point exceptions., 506
- forall, 263
- foralls, unused, 86
- forcing GHC-phase options, 182
- foreign, 263
- foreign export
  - with GHC, 464
- Foreign Function Interface
  - GHCi support, 17
- Foreign function interface, 461
- ForeignFunctionInterface
  - Language Extension, 461
- formatting dumps, 201
- framework-dirs
  - package specification, 180
- frameworks
  - package specification, 180
- fromInteger function, 506
- fromIntegral function, 506
- frontend plugins
  - using, 66
- FunctionalDependencies
  - Language Extension, 315
- GADTs
  - Language Extension, 275
- GADTSyntax
  - Language Extension, 271
- garbage collection

- compacting, 135
- garbage collector
  - options, 134
- GC sync time, measuring, 140
- GC threads, setting the number of, 137
- GeneralisedNewtypeDeriving
  - Language Extension, 297
- GeneralizedNewtypeDeriving
  - Language Extension, 297
- generations, number of, 136
- getArgs, behavior in GHCi, 50
- getProgName, behavior in GHCi, 50
- GHC backends, 180
- GHC code generators, 180
- GHC option
  - exclude-module=(file), 162
  - frontend (module), 66
  - help, 66
  - info, 67
  - interactive, 66
  - make, 66
  - mk-dll, 66
  - numeric-version, 67
  - print-libdir, 67
  - show-iface (file), 66, 151
  - show-options, 67
  - supported-extensions, 67
  - supported-languages, 67
  - version, 67
  - , 66
  - C, 66
  - D(symbol)[=(value)], 183
  - E, 66
  - F, 185
  - H (size), 73
  - I(dir), 183
  - L (dir), 188
  - M, 66
  - O, 88
  - O0, 88
  - O1, 88
  - O2, 88
  - Odph, 88
  - Rghc-timing, 73
  - S, 66
  - U(symbol), 183
  - V, 67
  - W, 75
  - Wall, 75
  - Wall-missed-specialisations, 77
  - Wamp, 78
  - Wcompat, 75
  - Wcpp-undef, 87
  - Wdeferred-out-of-scope-variables, 77
  - Wdeferred-type-errors, 76
  - Wdeprecated-flags, 79
  - Wdeprecations, 78
  - Wdodgy-exports, 79
  - Wdodgy-foreign-imports, 79
  - Wdodgy-imports, 79
  - Wduplicate-constraints, 80
  - Wduplicate-exports, 80
  - Wempty-enumerations, 79
  - Werror, 75
  - Weverything, 75
  - Whi-shadowing, 80
  - Widentities, 80
  - Wimplicit-prelude, 81
  - Wincomplete-patterns, 81
  - Wincomplete-record-updates, 81
  - Wincomplete-uni-patterns, 81
  - Winline-rule-shadowing, 87
  - Wmissed-specialisations, 77
  - Wmissing-export-lists, 82
  - Wmissing-exported-signatures, 83
  - Wmissing-exported-sigs, 83
  - Wmissing-fields, 82
  - Wmissing-home-modules, 87
  - Wmissing-import-lists, 82
  - Wmissing-local-signatures, 83
  - Wmissing-local-sigs, 83
  - Wmissing-methods, 82
  - Wmissing-monadfail-instances, 78
  - Wmissing-pattern-synonym-signatures, 83
  - Wmissing-signatures, 82
  - Wmonomorphism-restriction, 84
  - Wname-shadowing, 83
  - Wno-compat, 75
  - Wnoncanonical-monad-instances, 78
  - Wnoncanonical-monadfail-instances, 78
  - Wnoncanonical-monoid-instances, 78
  - Worphans, 83
  - Woverflowed-literals, 79
  - Woverlapping-patterns, 83
  - Wpartial-fields, 87
  - Wpartial-type-signatures, 77
  - Wredundant-constraints, 80
  - Wsafe, 459
  - Wsemigroup, 79
  - Wsimplifiable-class-constraints, 84
  - Wtabs, 84
  - Wtrustworthy-safe, 459
  - Wtype-defaults, 84
  - Wtyped-holes, 76
  - Wunbanged-strict-patterns, 87
  - Wunrecognised-pragmas, 77
  - Wunrecognised-warning-flags, 76



- Wunsafe, [459](#)
- Wunsupported-calling-conventions, [79](#)
- Wunsupported-llvm-version, [84](#)
- Wunticked-promoted-constructors, [84](#)
- Wunused-binds, [84](#)
- Wunused-do-bind, [86](#)
- Wunused-foralls, [86](#)
- Wunused-imports, [85](#)
- Wunused-local-binds, [85](#)
- Wunused-matches, [85](#)
- Wunused-pattern-binds, [85](#)
- Wunused-top-binds, [85](#)
- Wunused-type-patterns, [86](#)
- Wwarn, [76](#)
- Wwarnings-deprecations, [77](#)
- Wwrong-do-bind, [86](#)
- XEmptyDataDeriving, [286](#)
- XHexFloatLiterals, [242](#)
- c, [66](#), [187](#)
- clear-package-db, [169](#)
- cpp, [183](#)
- dcmm-lint, [202](#)
- dcore-lint, [202](#)
- ddump-asm, [200](#)
- ddump-asm-expanded, [200](#)
- ddump-asm-liveness, [200](#)
- ddump-asm-native, [200](#)
- ddump-asm-regalloc, [200](#)
- ddump-asm-regalloc-stages, [200](#)
- ddump-asm-stats, [200](#)
- ddump-bcos, [200](#)
- ddump-cmm, [200](#)
- ddump-cmm-caf, [199](#)
- ddump-cmm-cbe, [199](#)
- ddump-cmm-cfg, [199](#)
- ddump-cmm-cps, [200](#)
- ddump-cmm-from-stg, [199](#)
- ddump-cmm-info, [200](#)
- ddump-cmm-proc, [199](#)
- ddump-cmm-procmap, [199](#)
- ddump-cmm-raw, [199](#)
- ddump-cmm-sink, [199](#)
- ddump-cmm-sp, [199](#)
- ddump-cmm-split, [199](#)
- ddump-cmm-switch, [199](#)
- ddump-cmm-verbose, [199](#)
- ddump-core-stats, [198](#)
- ddump-cse, [198](#)
- ddump-deriv, [197](#)
- ddump-ds, [198](#)
- ddump-ec-trace, [197](#)
- ddump-foreign, [200](#)
- ddump-hi, [151](#)
- ddump-hi-diffs, [151](#)
- ddump-if-trace, [197](#)
- ddump-inlinings, [198](#)
- ddump-json, [196](#)
- ddump-llvm, [200](#)
- ddump-minimal-imports, [151](#)
- ddump-mod-cycles, [162](#)
- ddump-occur-anal, [199](#)
- ddump-opt-cmm, [200](#)
- ddump-parsed, [197](#)
- ddump-parsed-ast, [197](#)
- ddump-prep, [199](#)
- ddump-rn, [197](#)
- ddump-rn-ast, [197](#)
- ddump-rn-stats, [197](#)
- ddump-rn-trace, [197](#)
- ddump-rule-firings, [198](#)
- ddump-rule-rewrites, [198](#)
- ddump-rules, [198](#)
- ddump-simpl, [198](#)
- ddump-simpl-iterations, [198](#)
- ddump-simpl-stats, [198](#)
- ddump-spec, [198](#)
- ddump-splices, [197](#)
- ddump-stg, [199](#)
- ddump-str-signatures, [198](#)
- ddump-stranal, [198](#)
- ddump-tc, [197](#)
- ddump-tc-ast, [197](#)
- ddump-tc-trace, [197](#)
- ddump-timings, [196](#)
- ddump-to-file, [196](#)
- ddump-types, [197](#)
- ddump-vect, [198](#)
- ddump-vt-trace, [199](#)
- ddump-worker-wrapper, [198](#)
- debug, [189](#)
- dep-makefile {file}, [162](#)
- dep-suffix {suffix}, [162](#)
- dfaststring-stats, [196](#)
- dinitial-unique={s}, [202](#)
- distrust {pkg}, [457](#)
- distrust-all-packages, [457](#)
- dno-debug-output, [201](#)
- dppr-case-as-let, [201](#)
- dppr-cols={n}, [201](#)
- dppr-debug, [196](#)
- dppr-user-length, [201](#)
- dshow-passes, [196](#)
- dstg-lint, [202](#)
- dsuppress-all, [201](#)
- dsuppress-coercions, [201](#)
- dsuppress-idinfo, [201](#)
- dsuppress-module-prefixes, [201](#)
- dsuppress-stg-free-vars, [202](#)



- dsuppress-ticks, 201
- dsuppress-type-applications, 201
- dsuppress-type-signatures, 201
- dsuppress-unfoldings, 201
- dsuppress-uniques, 201
- dsuppress-var-kinds, 201
- dth-dec-file={file}, 197
- dumpdir {dir}, 150
- dunique-increment={i}, 202
- dverbose-core2core, 198
- dverbose-stg2stg, 199
- dylib-install-name {path}, 191
- dynamic, 188
- dynamic-too, 187
- dynload, 189
- e {expr}, 66
- eventlog, 190
- fPIC, 186
- fPIE, 186
- falignment-sanitisation, 202
- fasm, 186
- fbreak-on-error, 41
- fbreak-on-exception, 40
- fbyte-code, 186
- fcall-arity, 89
- fcase-folding, 89
- fcase-merge, 88
- fcatch-bottoms, 202
- fcmm-elim-common-blocks, 89
- fcmm-sink, 89
- fcpr-anal, 89
- fcross-module-specialise, 96
- fcse, 89
- fdefer-out-of-scope-variables, 77
- fdefer-type-errors, 76
- fdefer-typed-holes, 76
- fdiagnostics-color={always|auto|never},  
72
- fdiagnostics-show-caret, 73
- fdicts-cheap, 90
- fdicts-strict, 90
- fdmd-tx-dict-sel, 90
- fdo-eta-reduction, 90
- fdo-lambda-eta-expansion, 90
- feager-blackholing, 90
- fenable-rewrite-rules, 432
- ferror-spans, 73
- fexcess-precision, 90
- fexitification, 89
- fexpose-all-unfoldings, 90
- fexternal-interpreter, 58
- ffloat-in, 91
- fforce-recomp, 152
- ffull-laziness, 91
- ffun-to-thunk, 91
- fghci-hist-size={n}, 40
- fglasgow-exts, 235
- fhelptful-errors, 77
- fhide-source-paths, 70
- fhpc, 221
- fignore-asserts, 91
- fignore-hpc-changes, 152
- fignore-interface-pragmas, 91
- fignore-optim-changes, 152
- flate-dmd-anal, 91
- fliberate-case, 92
- fliberate-case-threshold={n}, 92
- fllvm, 186
- fllvm-fill-undef-with-garbage, 202
- fllvm-pass-vectors-in-regs, 92
- flocal-ghci-history, 42
- floopification, 92
- fmax-inline-alloc-size={n}, 92
- fmax-inline-memcpy-insns={n}, 92
- fmax-inline-memset-insns={n}, 92
- fmax-pmcheck-iterations={n}, 81
- fmax-relevant-binds={n}, 92
- fmax-simplifier-iterations={n}, 93
- fmax-uncovered-patterns={n}, 93
- fmax-valid-substitutions={n}, 92
- fmax-worker-args={n}, 93
- fno-code, 186
- fno-embed-manifest, 191
- fno-gen-manifest, 191
- fno-opt-coercion, 93
- fno-pre-inlining, 93
- fno-prof-auto, 208
- fno-prof-cafs, 208
- fno-prof-count-entries, 208
- fno-shared-implib, 191
- fno-state-hack, 93
- fobject-code, 186
- fomit-interface-pragmas, 93
- fomit-yields, 93
- foptimal-applicative-do, 250
- fpackage-trust, 459
- fpedantic-bottoms, 93
- fplugin-opt={module}:{args}, 475
- fplugin={module}, 475
- fprint-bind-result, 22
- fprint-equality-relations, 71
- fprint-expanded-synonyms, 71
- fprint-explicit-coercions, 71
- fprint-explicit-foralls, 70
- fprint-explicit-kinds, 71
- fprint-explicit-runtime-rep, 362
- fprint-explicit-runtime-reps, 71
- fprint-potential-instances, 70

- fprint-typechecker-elaboration, 72
- fprint-unicode-syntax, 70
- fprof-auto, 207
- fprof-auto-calls, 207
- fprof-auto-exported, 207
- fprof-auto-top, 207
- fprof-cafs, 207
- framework <name>, 187
- framework-path <dir>, 188
- fregs-graph, 94
- fregs-iterative, 94
- fshow-hole-constraints, 387
- fshow-loaded-modules, 18
- fshow-warning-groups, 76
- fsimpl-tick-factor=<n>, 94
- fsimplifier-phases=<n>, 94
- fsolve-constant-dicts, 96
- fspec-constr, 94
- fspec-constr-count=<n>, 95
- fspec-constr-keen, 95
- fspec-constr-threshold=<n>, 95
- fspecialise, 95
- fspecialise-aggressively, 96
- fstatic-argument-transformation, 96
- fstg-cse, 90
- fstrictness, 96
- fstrictness-before=<n>, 97
- funbox-small-strict-fields, 97
- funbox-strict-fields, 97
- funfolding-creation-threshold=<n>, 97
- funfolding-dict-discount=<n>, 98
- funfolding-fun-discount=<n>, 98
- funfolding-keenness-factor=<n>, 98
- funfolding-use-threshold=<n>, 98
- fvectorisation-avoidance, 98
- fvectorise, 98
- fwhole-archive-hs-libs, 192
- fwrite-interface, 186
- g, 485
- ghci-script, 57
- ghcversion-file <path to ghcversion.h>, 74
- global-package-db, 169
- g<n>, 485
- hcsuf <suffix>, 150
- hide-all-packages, 166
- hide-all-plugin-packages, 476
- hide-package <pkg>, 166
- hidir <dir>, 150
- hisuf <suffix>, 150
- i, 148
- ignore-dot-ghci, 57
- ignore-package <pkg>, 166
- include-pkg-deps, 162
- interactive-print <expr>, 32
- i<dir>[:<dir>]\*, 148
- j[<n>], 68
- keep-hc-file, 150
- keep-hc-files, 150
- keep-hi-files, 150
- keep-llvm-file, 150
- keep-llvm-files, 150
- keep-o-files, 151
- keep-s-file, 151
- keep-s-files, 151
- keep-tmp-files, 151
- l <lib>, 187
- main-is <thing>, 189
- msse2, 74
- msse4.2, 74
- no-auto-link-packages, 166
- no-global-package-db, 169
- no-hs-main, 189
- no-rtsopts-suggestions, 191
- no-user-package-db, 169
- o <file>, 149
- odir <dir>, 149
- ohi <file>, 149
- optF <option>, 182
- optL <option>, 182
- optP <option>, 182
- opta <option>, 182
- optc <option>, 182
- optdll <option>, 182
- opti <option>, 183
- optl <option>, 182
- optlc <option>, 182
- optlo <option>, 182
- optwindres <option>, 183
- osuf <suffix>, 150
- outputdir <dir>, 150
- package <name>, 187
- package <pkg>, 165
- package-db <file>, 169
- package-env <file>|<name>, 170
- package-id <unit-id>, 166
- pgmF <cmd>, 182
- pgmL <cmd>, 181
- pgmP <cmd>, 181
- pgma <cmd>, 182
- pgmc <cmd>, 181
- pgmdll <cmd>, 182
- pgmi <cmd>, 182
- pgml <cmd>, 182
- pgmlc <cmd>, 181
- pgmlibtool <cmd>, 182
- pgmlo <cmd>, 181
- pgms <cmd>, 182
- pgmwindres <cmd>, 182

- pie, 192
- plugin-package {pkg}, 476
- plugin-package-id {pkg-id}, 476
- prof, 207
- rdynamic, 192
- rtsopts={none|some|all}, 190
- shared, 188
- split-objs, 188
- split-sections, 188
- static, 188
- staticlib, 188
- stubdir {dir}, 150
- this-unit-id {unit-id}, 166
- threaded, 189
- ticky, 225
- tmpdir {dir}, 151
- trust {pkg}, 457
- user-package-db, 169
- v, 69
- v(n), 69
- w, 75
- with-rtsopts={opts}, 190
- x {suffix}, 69
- GHC vs the Haskell standards, 501
- GHC, using, 63
- GHC\_PACKAGE\_PATH, 168, 169, 172, 173
- GHCi, 17, 66
- GHCi command
  - ., 47
  - {command}, 54
  - ?, 47
  - :abandon, 44
  - :add, 44
  - :all-types, 44
  - :back, 44
  - :break, 44
  - :browse, 44
  - :cd, 45
  - :cmd, 45
  - :complete, 45
  - :continue, 46
  - :ctags, 46
  - :def, 46
  - :delete, 47
  - :edit, 47
  - :etags, 47
  - :force, 47
  - :forward, 47
  - :help, 47
  - :history, 47
  - :info, 47
  - :issafe, 48
  - :kind, 48
  - :list, 48
  - :list [{module}], 48
  - :load, 48
  - :loc-at, 49
  - :main, 49
  - :module, 49
  - :print, 50
  - :quit, 50
  - :reload, 50
  - :run, 50
  - :script, 50
  - :set, 50
  - :set +c, 54
  - :set +m, 54
  - :set +r, 54
  - :set +s, 54
  - :set +t, 54
  - :set args, 50
  - :set editor, 50
  - :set prog, 50
  - :set prompt, 50
  - :set prompt-cont, 51
  - :set prompt-cont-function, 51
  - :set prompt-function, 51
  - :set stop, 51
  - :seti, 51
  - :show, 52
  - :show bindings, 52
  - :show breaks, 52
  - :show context, 52
  - :show imports, 52
  - :show language, 52
  - :show modules, 52
  - :show packages, 52
  - :show paths, 52
  - :showi language, 52
  - :sprint, 52
  - :step, 52
  - :steplocal, 52
  - :stepmodule, 52
  - :trace, 52
  - :type, 53
  - :type +d, 53
  - :type +v, 53
  - :type-at, 53
  - :undef, 53
  - :unset, 53
  - :uses, 53
  - import, 49
- GHCi prompt
  - setting, 50
- GHCi prompt function
  - setting, 51
- GHCRS, 131, 132, 190
- GHCRS environment variable, 132

- Glasgow Haskell mailing lists, 5
- group, 252
- haddock-html
  - package specification, 180
- haddock-interfaces
  - package specification, 180
- Happy, 491
- happy parser generator, 491
- Haskell Program Coverage, 220
- Haskell standards vs GHC, 501
- heap profiles, 216
- heap size, factor, 136
- heap size, grace, 139
- heap size, maximum, 138
- heap size, suggested, 137
- heap space, using less, 231
- heap, minimum free, 138
- help options, 66
- hidden-modules
  - package specification, 178
- HOME, 44, 45
- homepage
  - package specification, 178
- hooks
  - RTS, 132
- hp2ps, 216
- hp2ps command line option
  - , 217
  - b, 216
  - c, 217
  - d, 216
  - e(float)[in|mm|pt], 216
  - g, 217
  - l, 217
  - m(int), 217
  - p, 217
  - s, 217
  - t(float), 217
  - y, 217
- hpc, 220
- HPCTIXFILE, 221, 225
- hs-libraries
  - package specification, 179
- hs\_exit, 505
- hs\_init, 505
- hsc2hs, 491
- Hugs, 17
- hugs-options
  - package specification, 179
- id
  - package specification, 178
- idle GC, 137
- implicit parameters, 263
- implicit prelude, warning, 81
- ImplicitParams
  - Language Extension, 377
- import
  - GHCi command, 49
- import lists, missing, 82
- import-dirs
  - package specification, 179
- importing, “hi-boot” files, 153
- imports, unused, 85
- ImpredicativeTypes
  - Language Extension, 384
- improvement, code, 87
- in GHCi
  - Repeating last command, 47
- include-dirs
  - package specification, 179
- includes
  - package specification, 179
- INCOHERENT, 432
- IncoherentInstances
  - Language Extension, 324
- incomplete patterns, warning, 81
- incomplete record updates, warning, 81
- INLINE, 423
- inlining, controlling, 97, 98
- InstanceSigs
  - Language Extension, 327
- Int
  - size of, 506
- interactive, 17
- Interactive classes, 31
- interactive mode, 66
- interface files, 147
- interface files, finding them, 148
- interface files, options, 151
- interfacing with native code, 461
- intermediate files, saving, 150
- intermediate passes, output, 196
- interpreter, 17
- InterruptibleFFI
  - Language Extension, 462
- invoking
  - GHCi, 42
- it variable, 29
- kind heterogeneous
  - Type equality constraints, 365
- KindSignatures
  - Language Extension, 369
- LambdaCase
  - Language Extension, 259

- LANGUAGE
  - pragma, 420
- language
  - option, 233
- Language Extension
  - AllowAmbiguousTypes, 368
  - ApplicativeDo, 248
  - Arrows, 404
  - BangPatterns, 411
  - BinaryLiterals, 242
  - CapiFFI, 463
  - ConstrainedClassMethods, 313
  - ConstraintKinds, 366
  - CPP, 183
  - DataKinds, 349
  - DatatypeContexts, 264
  - DefaultSignatures, 314
  - DeriveAnyClass, 302
  - DeriveDataTypeable, 295
  - DeriveFoldable, 292
  - DeriveFunctor, 289
  - DeriveGeneric, 440
  - DeriveLift, 296
  - DeriveTraversable, 294
  - DerivingStrategies, 304
  - DisambiguateRecordFields, 277
  - DuplicateRecordFields, 278
  - EmptyCase, 259
  - EmptyDataDecls, 264
  - ExistentialQuantification, 267
  - ExplicitForAll, 367
  - ExplicitNamespaces, 262
  - ExtendedDefaultRules, 30
  - FlexibleContexts, 312
  - FlexibleInstances, 321
  - ForeignFunctionInterface, 461
  - FunctionalDependencies, 315
  - GADTs, 275
  - GADTSyntax, 271
  - GeneralisedNewtypeDeriving, 297
  - GeneralizedNewtypeDeriving, 297
  - ImplicitParams, 377
  - ImpredicativeTypes, 384
  - IncoherentInstances, 324
  - InstanceSigs, 327
  - InterruptibleFFI, 462
  - KindSignatures, 369
  - LambdaCase, 259
  - LiberalTypeSynonyms, 266
  - MagicHash, 241
  - MonadComprehensions, 253
  - MonadFailDesugaring, 256
  - MonoLocalBinds, 374
  - MultiParamTypeClasses, 312
  - MultiWayIf, 260
  - NamedFieldPuns, 280
  - NamedWildCards, 389
  - NegativeLiterals, 242
  - NoImplicitPrelude, 256
  - NoMonomorphismRestriction, 374
  - NoPatternGuards, 243
  - NoTraditionalRecordSyntax, 277
  - NPlusKPatterns, 245
  - NullaryTypeClasses, 315
  - NumDecimals, 242
  - OverlappingInstances, 324
  - OverloadedLabels, 329
  - OverloadedLists, 331
  - OverloadedStrings, 328
  - PackageImports, 261
  - ParallelListComp, 251
  - PartialTypeSignatures, 387
  - PatternSynonyms, 305
  - PolyKinds, 352
  - PostfixOperators, 258
  - QuasiQuotes, 402
  - Rank2Types, 380
  - RankNTypes, 380
  - RebindableSyntax, 256
  - RecordWildCards, 281
  - RecursiveDo, 245
  - RoleAnnotations, 444
  - Safe, 458
  - ScopedTypeVariables, 370
  - StandaloneDeriving, 287
  - StaticPointers, 418
  - Strict, 412
  - StrictData, 412
  - TemplateHaskell, 394
  - TemplateHaskellQuotes, 395
  - TransformListComp, 252
  - Trustworthy, 458
  - TupleSections, 258
  - TypeApplications, 375
  - TypeFamilies, 334
  - TypeFamilyDependencies, 347
  - TypeInType, 352
  - TypeOperators, 265
  - TypeSynonymInstances, 320
  - UnboxedSums, 239
  - UnboxedTuples, 238
  - UndecidableInstances, 322
  - UndecidableSuperClasses, 333
  - UnicodeSyntax, 240
  - Unsafe, 459
  - ViewPatterns, 243
- language, GHC extensions, 233
- Latin-1, 147

- ld options, [187](#)
- ld-options
  - package specification, [180](#)
- LD\_LIBRARY\_PATH, [43](#), [195](#)
- levity polymorphism, [361](#)
- lhs file extension, [65](#)
- libdir, [67](#)
- LiberalTypeSynonyms
  - Language Extension, [266](#)
- libraries
  - with GHCi, [43](#)
- library-dirs
  - package specification, [179](#)
- license-file
  - package specification, [178](#)
- LINE
  - pragma, [426](#)
- linker options, [187](#)
- linking Haskell libraries with foreign code, [189](#)
- lint, [202](#)
- list comprehensions
  - ambiguity with quasi-quotes, [403](#)
  - generalised, [252](#)
  - parallel, [251](#)
- LLVM code generator, [180](#)
- machine-specific options, [74](#)
- MagicHash
  - Language Extension, [241](#)
- mailing lists, Glasgow Haskell, [5](#)
- maintainer
  - package specification, [178](#)
- make
  - building programs with, [160](#)
- make and recompilation, [147](#)
- make mode
  - of GHC, [66](#)
- Makefile dependencies, [161](#)
- Makefiles
  - avoiding, [67](#)
- MallocFailHook (C function), [133](#)
- matches, unused, [85](#)
- mdo, [263](#)
- memory, using less heap, [231](#)
- methods, missing, [82](#)
- MFP, [79](#)
- MIN\_VERSION\_GLASGOW\_HASKELL, [184](#)
- MINIMAL, [422](#)
- miscellaneous flags, [74](#)
- missing export lists, warning, [82](#)
- missing fields, warning, [82](#)
- missing import lists, warning, [82](#)
- missing methods, warning, [82](#)
- mode
  - options, [65](#)
- module system, recursion, [152](#)
- modules
  - and filenames, [18](#)
- monad comprehensions, [253](#)
- MonadComprehensions
  - Language Extension, [253](#)
- MonadFail Proposal, [79](#)
- MonadFailDesugaring
  - Language Extension, [256](#)
- MonoLocalBinds
  - Language Extension, [374](#)
- monomorphism restriction, warning, [84](#)
- multiline input
  - in GHCi, [54](#)
- MultiParamTypeClasses
  - Language Extension, [312](#)
- MultiWayIf
  - Language Extension, [260](#)
- name
  - package specification, [178](#)
- NamedFieldPuns
  - Language Extension, [280](#)
- NamedWildCards
  - Language Extension, [389](#)
- native code generator, [180](#)
- NegativeLiterals
  - Language Extension, [242](#)
- NoImplicitPrelude
  - Language Extension, [256](#)
- NOINLINE, [425](#)
- nominal
  - role, [443](#)
- NoMonomorphismRestriction
  - Language Extension, [374](#)
- NoPatternGuards
  - Language Extension, [243](#)
- NOTINLINE, [425](#)
- NoTraditionalRecordSyntax
  - Language Extension, [277](#)
- NOUNPACK, [430](#)
- NPlusKPatterns
  - Language Extension, [245](#)
- NullaryTypeClasses
  - Language Extension, [315](#)
- NUMA, enabling in the runtime, [139](#)
- NumDecimals
  - Language Extension, [242](#)
- object files, [147](#)
- old generation, size, [135](#)
- optimisation, [87](#)



- optimise
  - aggressively, 88
  - DPH, 88
  - normally, 88
- optimization
  - and GHCi, 58
- options
  - for profiling, 207
  - GHCi, 54
  - language, 233
- OPTIONS\_GHC, 421
- orphan instances, warning, 83
- orphan rules, warning, 83
- OutOfHeapHook (C function), 132
- output-directing options, 149
- OVERLAPPABLE, 432
- OVERLAPPING, 432
- overlapping patterns, warning, 83
- OverlappingInstances
  - Language Extension, 324
- OVERLAPS, 432
- OverloadedLabels
  - Language Extension, 329
- OverloadedLists
  - Language Extension, 331
- OverloadedStrings
  - Language Extension, 328
- overloading, death to, 228, 426, 429
- package environments, 170
- package trust, 457
- package-url
  - package specification, 178
- PackageImports
  - Language Extension, 261
- packages, 163
  - building, 175
  - management, 172
  - using, 164
  - with GHCi, 43
- parallel list comprehensions, 251
- parallelism, 99, 447
- ParallelListComp
  - Language Extension, 251
- parser generator for Haskell, 491
- PartialTypeSignatures
  - Language Extension, 387
- PATH, 169, 186
- patterns, incomplete, 81
- patterns, overlapping, 83
- PatternSynonyms
  - Language Extension, 305
- phantom
  - role, 443
- platform-specific options, 74
- PolyKinds
  - Language Extension, 352
- PostfixOperators
  - Language Extension, 258
- postscript, from heap profiles, 216
- pragma, 420
  - ANN, 473
  - COLUMN, 426
  - INCOHERENT, 432
  - INLINE, 423
  - LANGUAGE, 420
  - LINE, 426
  - MINIMAL, 422
  - OPTIONS\_GHC, 421
  - OVERLAPPABLE, 432
  - OVERLAPPING, 432
  - OVERLAPS, 432
  - RULES, 432
  - SOURCE, 430
- pragma, SPECIALIZE, 426
- pre-processing: cpp, 183
- pre-processing: custom, 185
- pre-processor options, 185
- problems, 483
- problems running your program, 484
- problems with the compiler, 483
- proc, 263
- profiling, 203
  - options, 207
  - ticky ticky, 145
  - with Template Haskell, 401
- promoted constructor, warning, 84
- prompt
  - GHCi, 17
- quasi-quotation, 263
- Quasi-quotes, 263
- quasi-quotes
  - ambiguity with list comprehensions, 403
- QuasiQuotes
  - Language Extension, 402
- Rank2Types
  - Language Extension, 380
- RankNTypes
  - Language Extension, 380
- RebindableSyntax
  - Language Extension, 256
- recompilation checker, 147, 152
- record updates, incomplete, 81
- RecordWildCards
  - Language Extension, 281
- recursion, between modules, 152

- RecursiveDo
  - Language Extension, 245
- redirecting compilation output, 149
- redundant constraints, warning, 80
- reexported-modules
  - reexport specification, 178
- Repeating last command
  - in GHCi, 47
- reporting bugs, 6
- representational
  - role, 443
- rewrite rules, 432
- RoleAnnotations
  - Language Extension, 444
- roles, 442
- RPATH, 195
- RTS, 146
- RTS behaviour, changing, 132
- RTS hooks, 132
- RTS option
  - generate-crash-dumps, 133
  - generate-stack-traces=<yes|no>, 133
  - info, 146
  - install-seh-handlers=<yes|no>, 133
  - install-signal-handlers=<yes|no>, 133
  - long-gc-sync, 139
  - long-gc-sync=<seconds>, 139
  - machine-readable, 140
  - numa, 139
  - numa=<mask>, 139
  - A {size}, 134
  - AL {size}, 134
  - B, 144
  - C {s}, 99
  - D {x}, 144
  - F {factor}, 135
  - G {generations}, 136
  - H [{size}], 137
  - I {seconds}, 137
  - K {size}, 138
  - L {num}, 214
  - L {n}, 143
  - M {size}, 138
  - Mgrace={size}, 139
  - N {x}, 100
  - O {size}, 135
  - P, 208
  - R {size}, 214
  - S [{file}], 140
  - T, 140
  - V {secs}, 208
  - Z, 145
  - c, 135
  - c {n}, 135
  - h, 143, 212
  - hT, 143, 212
  - hb, 213
  - hc, 212
  - hd, 212
  - hm, 212
  - hr, 213
  - hy, 213
  - i {secs}, 214
  - kb {size}, 138
  - kc {size}, 138
  - ki {size}, 137
  - l, 213
  - l {flags}, 143
  - m {n}, 138
  - maxN {x}, 100
  - n {size}, 135
  - p, 208
  - pa, 208
  - pj, 208
  - po {stem}, 208
  - qa, 100
  - qb {gen}, 136
  - qg {gen}, 136
  - qm, 101
  - qn {x}, 136
  - r {file}, 145
  - s [{file}], 140
  - t [{file}], 140
  - v [{flags}], 144
  - xc, 145, 209
  - xm {address}, 133
  - xq {size}, 134
  - xt, 214
- RTS options, 131
  - concurrent, 99
  - from the environment, 132
  - garbage collection, 134
- RTS options, hacking/debugging, 144
- RTS options, setting, 131
- RULES pragma, 432
- runghc, 61
- running, compiled program, 131
- RUNPATH, 195
- runtime control of Haskell programs, 131
- Safe
  - Language Extension, 458
- safe compilation, 459
- safe haskell, 449
- Safe Haskell flags, 458
- safe haskell trust, 455
- safe haskell uses, 450
- safe imports, 455



- safe inference, 458
- safe language, 452
- sanity-checking options, 74
- ScopedTypeVariables
  - Language Extension, 370
- search path, 148
  - source code, 148
- secure haskell, 450
- semigroup
  - warning, 79
- separate compilation, 67, 147
- shadowing
  - interface files, 80
- shadowing, warning, 83
- Shared libraries
  - using, 192
- shell commands
  - in GHCi, 54
- Show class, 30
- signature files
  - Backpack
  - hsig files, 154
- simple: stack trace
  - in GHCi, 32
- simplifiable class constraints, warning, 84
- single : -osuf
  - using with profiling, 401
- smaller programs, how to produce, 231
- SMP, 99, 448
- SOURCE, 430
- source annotations, 473
- source-file options, 64
- space-leaks, avoiding, 231
- SPECIALIZE pragma, 228, 426, 429
- specifying your own main function, 189
- SQL, 252
- stability
  - package specification, 178
- stack
  - chunk buffer size, 138
  - chunk size, 138
- stack, initial size, 137
- stack, maximum size, 138
- StackOverflowHook (C function), 132
- StandaloneDeriving
  - Language Extension, 287
- startup
  - files, GHCi, 56, 57
- statements
  - in GHCi, 22
- static
  - options, 55, 65
- Static pointers, 418
- StaticPointers
  - Language Extension, 418
- Strict
  - Language Extension, 412
- strict constructor fields, 97
- strict haskell, 410
- StrictData
  - Language Extension, 412
- string gaps vs -cpp., 185
- structure, command-line, 64
- suffixes, file, 65
- suppression
  - of unwanted dump output, 201
- tabs, warning, 84
- Template Haskell, 263
- TemplateHaskell
  - Language Extension, 394
- TemplateHaskellQuotes
  - Language Extension, 395
- temporary files
  - keeping, 150, 151
  - redirecting, 151
- ticky ticky profiling, 145
- ticky-ticky profiling, 225
- time profile, 208
- TMPDIR, 151
- TMPDIR environment variable, 151
- tracing, 143
- TransformListComp
  - Language Extension, 252
- trust, 455
- trust check, 455, 456
- trusted
  - package specification, 179
- Trustworthy
  - Language Extension, 458
- trustworthy, 457
- TupleSections
  - Language Extension, 258
- TYPE, 361
- Type defaulting
  - in GHCi, 30
- Type equality constraints
  - kind heterogeneous, 365
- type patterns, unused, 86
- type signatures, missing, 82, 83
- type signatures, missing, pattern synonyms, 83
- TypeApplications
  - Language Extension, 375
- TypeFamilies
  - Language Extension, 334
- TypeFamilyDependencies
  - Language Extension, 347

- TypeInType
  - Language Extension, [352](#)
- TypeOperators
  - Language Extension, [265](#)
- TypeSynonymInstances
  - Language Extension, [320](#)
- UnboxedSums
  - Language Extension, [239](#)
- UnboxedTuples
  - Language Extension, [238](#)
- UndecidableInstances
  - Language Extension, [322](#)
- UndecidableSuperClasses
  - Language Extension, [333](#)
- unfolding, controlling, [97](#), [98](#)
- Unicode, [147](#)
- UnicodeSyntax
  - Language Extension, [240](#)
- UNPACK, [429](#)
- unregisterised compilation, [181](#)
- Unsafe
  - Language Extension, [459](#)
- unused binds, warning, [84](#), [85](#)
- unused do binding, warning, [86](#)
- unused forall, warning, [86](#)
- unused imports, warning, [85](#)
- unused matches, warning, [85](#)
- unused type patterns, warning, [86](#)
- using GHC, [63](#)
- UTF-8, [147](#)
- utilities, Haskell, [491](#)
- verbosity options, [69](#)
- version
  - package specification, [178](#)
- version, of ghc, [6](#)
- ViewPatterns
  - Language Extension, [243](#)
- WARNING, [421](#)
- warnings, [74](#)
  - deprecations, [77](#)
- windres, [191](#)
- Yacc for Haskell, [491](#)