The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.10.0.20141222

COLLABORATORS

	TITLE : The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.10.0.20141222		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	The GHC Team	December 22, 2014	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

The Glasgow Haskell Compiler License

Chapter 1

Introduction to GHC

This is a guide to using the Glasgow Haskell Compiler (GHC): an interactive and batch compilation system for the Haskell 98 language.

GHC has two main components: an interactive Haskell interpreter (also known as GHCi), described in Chapter 2, and a batch compiler, described throughout Chapter 4. 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 Chapter 7.

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 Chapter 5 for more details.

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

1.1 Obtaining GHC

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

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

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

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

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

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

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

list email address: glasgow-haskell-users@haskell.org

subscribe at: `http://www.haskell.org/mailman/listinfo/glasgow-haskell-users`.

admin email address: glasgow-haskell-users-admin@haskell.org

list archives: `http://www.haskell.org/pipermail/glasgow-haskell-users/`

ghc-devs: The hardcore GHC developers hang out here.

list email address: ghc-devs@haskell.org

subscribe at: `http://www.haskell.org/mailman/listinfo/ghc-devs`.

admin email address: ghc-devs-admin@haskell.org

list archives: `http://www.haskell.org/pipermail/ghc-devs/`

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

Some Haskell-related discussion also takes place in the Usenet newsgroup `comp.lang.functional`.

1.3 Reporting bugs in GHC

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

1.4 GHC version numbering policy

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

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

The value of `__GLASGOW_HASKELL__` (see Section 4.12.3) for a major release *x.y.z* is the integer *xyy* (if *y* is a single digit, then a leading zero is added, so for example in version 6.8.2 of GHC we would have `__GLASGOW_HASKELL__==608`).

Stable snapshots 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`.

The value of `__GLASGOW_HASKELL__` for a snapshot release is the integer *xyy*. You should never write any conditional code which tests for this value, however: since interfaces change on a day-to-day basis, and we don't have finer granularity in the values of `__GLASGOW_HASKELL__`, you should only conditionally compile using predicates which test whether `__GLASGOW_HASKELL__` is equal to, later than, or earlier than a given major release.

Unstable snapshots 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 *xyy*. You should never write any conditional code which tests for this value, however: since interfaces change on a day-to-day basis, and we don't have finer granularity in the values of `__GLASGOW_HASKELL__`, you should only conditionally compile using predicates which test whether `__GLASGOW_HASKELL__` is equal to, later than, or earlier than a given major release.

The version number of your copy of GHC can be found by invoking `ghc` with the `--version` flag (see Section 4.6).

1.5 Release notes for version 7.10.1

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 7.8 branch.

1.5.1 Highlights

The highlights, since the 7.8 branch, are:

- TODO FIXME

1.5.2 Full details

1.5.2.1 Language

- There is a new extension, StaticPointers, which allows you to create pointers to expressions which remain valid across processes. This is useful for referencing higher-order values in distributed systems. The pointers are created with a new keyword `static` as in `x =static ("abc" ++ "123") ::StaticPtr String`. All processes which dereference `x` get the same result, that is, the body of the static form.
- Added support for binary integer literals
- Simplified rules for implicit quantification. In previous versions of GHC, it was possible to use the `=>` arrow to quantify over type variables in `data` and `type` declarations without a `forall` quantifier. For example, `data Fun =Fun (Ord a => a -> b)` was identical to `data Fun =Fun (forall a b.Ord a => a -> b)`, while `data Fun =Fun (a -> b)` caused a not-in-scope error. This implicit quantification is now deprecated, and variables in higher-rank constructors should be quantified with `forall` regardless of whether a class context is present or not. GHC 7.10 raises a warning (controlled by `-fwarn-context-quantification`, enabled by default) and GHC 7.12 will raise an error. See examples in GHC documentation.

  The change also applies to Template Haskell splices such as `[t|Ord a => a|]`, which should be written as `[t|forall a.Ord a => a|]`.
- Instance contexts inferred while processing `deriving` directives attached to `data` and `newtype` declarations now forbid equality constraints. This is a regression in obscure cases, but it will yield better error messages in more common cases. Users caught by the regression can simply use standalone-deriving, where you specify the context yourself.

1.5.2.2 Compiler

- GHC now checks that all the language extensions required for the inferred type signatures are explicitly enabled. This means that if any of the type signatures inferred in your program requires some language extension you will need to enable it. The motivation is that adding a missing type signature inferred by GHC should yield a program that typechecks. Previously this was not the case.

  This is a breaking change. Code that used to compile in the past might fail with an error message requiring some particular language extension (most likely `-XTypeFamilies`, `-XGADTs` or `-XFlexibleContexts`).
- `-fwarn-tabs` warning flag is turned on by default with this release of GHC. It can be suppressed either by using `GHC_OPTIONS` pragma or by specifying `-fno-warn-tabs` flag.
- A new warning flag, `-fwarn-trustworthy-safe` has been added and is turned on with `-Wall`. It warns when a module that is compiled with `-XTrustworthy` is actually infered as an `-XSafe` module. This lets the module author know that they can tighten their Safe Haskell bounds if desired.

- The `-fwarn-safe` and `-fwarn-unsafe` that warn if a module was infered as Safe or Unsafe have been improved to work with all Safe Haskell module types. Previously, they only worked for unmarked modules where the compiler was infering the modules Safe Haskell type. They now work even for modules marked as `-XTrustworthy` or `-XUnsafe`. This is useful either to have GHC check your assumptions, or to generate a list of reasons easily why a module is regarded as Unsafe.

  For many use cases, the new `-fwarn-trustworthy-safe` flag is better suited than either of these two.

- `-ddump-simpl-phases` and `-ddump-core-pipeline` flags have been removed.

- Many more options have learned to respect the `-ddump-to-file`. For example you can use `-ddump-to-file` with `-ddump-splices` to produce a `.dump-splices file` for each file that uses Template Haskell. This should be much easier to understand on a larger project than having everything being dumped to stdout.

- Compiler plugins (with the `-fplugin` flag) may now modify the behaviour of the constraint solver, to add new functionality to GHC's typechecker. See Section 9.3.4 for more details.

- A new warning flag, `-fwarn-missing-exported-sigs` has been added. The behavior is similar to `-fwarn-missing-signatures` but GHC will only flag exported values. This flag takes precedence over `-fwarn-missing-signatures` so it can be used in conjunction with `-Wall`.

1.5.2.3 GHCi

- TODO FIXME

1.5.2.4 Template Haskell

- Added support for generating LINE pragma declarations (Section 7.22.7).

- The type `Pred` (which stores a type constraint) is now a synonym for `Type`, in order to work with the `ConstraintKinds` extension. This is a breaking change and may require some rewriting of Template Haskell code.

- Pattern splices now work.

- `reifyInstances` now treats unbound type variables as univerally quantified, allowing lookup of, say, the instance for `Eq [a]`.

- More kind annotations appear in reified types, in order to disambiguate types that would otherwise be ambiguous in the presence of `PolyKinds`. In particular, all reified `TyVarBndrs` are now `KindedTVs`. (This does not affect Template Haskell quotations, just calls to `reify`.)

- Various features unsupported in quotations were previously silently ignored. These now cause errors.

- `Lift` instances were added for many more types: all of the `IntXX` and `WordXX` types, `Ratio a`, `()`, `Float`, and `Double`.

- All Template Haskell datatypes now have `Generic` and `Ord` instances.

- `Ppr` instances were added for `Lit` and `Loc`.

- Two new declaration forms are now supported: standalone-deriving declarations and generic method signatures (written using `default` in a class). This means an expansion to the `Dec` type.

- Template Haskell is now more pedantic about splicing in bogus variable names, like those containing whitespace. If you use bogus names in your Template Haskell code, this may break your program.

1.5.2.5 Runtime system

- The linker API is now thread-safe. The main user-facing impact of this change is that you must now call `initLinker` before calling `loadObj` or any of the other linker APIs.

1.5.2.6 Build system

- `ghc-pkg` now respects `--user` and `--global` when modifying packages (e.g. changing exposed/trust flag or unregistering). Previously, `ghc-pkg` would ignore these flags and modify whichever package it found first on the database stack. To recover the old behavior, simply omit these flags.
- `ghc-pkg` accepts a `--user-package-db` flag which allows a user to override the location of the user package database. Unlike databases specified using `--package-db`, a user package database configured this way respects the `--user` flag.

1.5.2.7 Package system

- TODO: cover module renaming, thinning, re-export etc
- ghc-pkg (and ghc) have dropped support for single-file style package databases. Since version 6.12, ghc-pkg has defaulted to a new database format (using a directory of files, one per package plus a binary cache).

  This change will not affect programs and scripts that use `ghc-pkg init` to create package databases.

  This will affect scripts that create package databases using tricks like

  ```
  echo "[]" > package.conf
  ```

  Such scripts will need to be modified to use `ghc-pkg init`, and to delete databases by directory removal, rather than simple file delete.

1.5.3 Libraries

1.5.3.1 array

- Version number XXXXX (was 0.5.0.0)

1.5.3.2 base

- Version number XXXXX (was 4.7.0.0)
- GHC has had its internal Unicode database for parsing updated to the Unicode 7.0 standard.
- Attempting to access a portion of the result of `System.IO.hGetContents` that was not yet read when the handle was closed now throws an exception. Previously, a lazy read from a closed handle would simply end the result string, leading to silent or delayed failures.

1.5.3.3 bin-package-db

- This is an internal package, and should not be used.

1.5.3.4 binary

- Version number XXXXX (was 0.7.1.0)

1.5.3.5 bytestring

- Version number XXXXX (was 0.10.4.0)

1.5.3.6 Cabal

- Version number XXXXX (was 1.18.1.3)

1.5.3.7 containers

- Version number XXXXX (was 0.5.4.0)

1.5.3.8 deepseq

- Version number XXXXX (was 1.3.0.2)

1.5.3.9 directory

- Version number XXXXX (was 1.2.0.2)

1.5.3.10 filepath

- Version number XXXXX (was 1.3.0.2)

1.5.3.11 ghc

- Many internal functions in GHC related to package IDs have been renamed to refer to package keys, e.g. `PackageId` is now `PackageKey`, the wired-in names such as `primPackageId` are now `primPackageKey`, etc. This reflects a distinction that we are now making: a package ID is, as before, the user-visible ID from Cabal `foo-1.0`; a package key is now a compiler-internal entity used for generating linking symbols, and may not correspond at all to the package ID. In particular, there may be multiple package keys per package ID.
- The ghc library no longer depends on the Cabal library. This means that users of the ghc library are no longer forced to use the same version of Cabal as ghc did. It also means that Cabal is freed up to be able to depend on packages that ghc does not want to depend on (which for example may enable improvements to Cabal's parsing infrastructure).

1.5.3.12 ghc-prim

- Version number XXXXX (was 0.3.1.0)

1.5.3.13 haskell98

- Version number XXXXX (was 2.0.0.3)

1.5.3.14 haskell2010

- Version number XXXXX (was 1.1.1.1)

1.5.3.15 hoopl

- Version number XXXXX (was 3.10.0.0)

1.5.3.16 hpc

- Version number XXXXX (was 0.6.0.1)

1.5.3.17 integer-gmp

- Version number XXXXX (was 0.5.1.0)

1.5.3.18 old-locale

- Version number XXXXX (was 1.0.0.6)

1.5.3.19 old-time

- Version number XXXXX (was 1.1.0.2)

1.5.3.20 process

- Version number XXXXX (was 1.2.0.0)

1.5.3.21 template-haskell

- Version number XXXXX (was 2.9.0.0)

1.5.3.22 time

- Version number XXXXX (was 1.4.1)

1.5.3.23 unix

- Version number XXXXX (was 2.7.0.0)

1.5.3.24 Win32

- Version number XXXXX (was 2.3.0.1)

1.5.4 Known bugs

- TODO FIXME

Chapter 2

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 Section 2.5).

2.1 Introduction to GHCi

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

```
$ ghci
GHCi, version 6.12.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
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 `:?` 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 Section 2.7.

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.

2.2 Loading source files

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

[1] The 'i' stands for "Interactive"
[2] except `foreign export`, at the moment

```
main = print (fac 20)

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

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

```
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` 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 Section 2.4.5). 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` command (hint: `:load` 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.

2.2.1 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`, or specify it when you invoke `ghci`, you can give a filename rather than a module name. This filename is loaded if it exists, and it may contain any module you like. This is particularly convenient if you have several `Main` modules in the same directory and you can't call them all `Main.hs`.

The search path for finding source files is specified with the `-i` option on the GHCi command line, like so:

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

or it can be set using the `:set` command from within GHCi (see Section 2.8.2)[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.

2.2.2 Making changes and recompilation

If you make some changes to the source code and want GHCi to recompile the program, give the `:reload` 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 Section 4.7.8).

[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*.

[4] Note that in GHCi, and `--make` mode, the `-i` option is used to specify the search path for *source* files, whereas in standard batch-compilation mode the `-i` option is used to specify the search path for interface files, see Section 4.7.3.

2.3 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`, 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 & C:

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

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

```
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.
*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` 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` 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`, only `:load`:

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

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 Section 2.4.5). 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`, 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` option (see Section 2.10).

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.

2.4 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
```

2.4.1 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"
```

2.4.2 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.

If `-fprint-bind-result` 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.

Hint: you can also use `let`-statements to define functions at the prompt:

```
Prelude> let 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 braces and semicolons instead of layout:

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

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

Such multiline commands can be used with any GHCi command, and the lines between `:{` and `:}` are simply merged into a single line for interpretation. That implies that each such group must form a single valid command when merged, and that no layout rule is used. The main purpose of multiline commands is not to replace module loading but to make definitions in .ghci-files (see Section 2.9) 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` or `:reload` command, at which time they will be simply lost. However, they do survive a change of context with `:module`: the temporary bindings just move to the new location.

HINT: To get a list of the bindings currently in scope, use the `:show bindings` 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]
```

2.4.3 Multiline input

Apart from the `:{ ... :}` syntax for multi-line input mentioned above, GHCi also has a multiline mode, enabled by `: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, as usual:

```
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>
```

2.4.4 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 or type-family instance *replaces* any earlier instance with an identical head or left hand side (respectively). (See Section 7.7.)

2.4.5 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`, `:add`, and `:reload` commands (Section 2.4.5.1).
- The `import` declaration (Section 2.4.5.2).
- The `:module` command (Section 2.4.5.3).

The command `:show imports` 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 `".

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

The `:load`, `:add`, and `:reload` commands (Section 2.2 and Section 2.3) 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` 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`:

```
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`, `:add`, or `:reload`. It can also be removed by `:module` as with normal imports.

2.4.5.2 Controlling what is in scope with `import`

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

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

```
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`:

```
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.

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

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

- `:module` 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.

2.4.5.4 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 flag `-fno-implicit-import-qualified`.

2.4.5.5 `:module` and `:load`

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

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

2.4.6 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` command. This runs whatever `main` is in scope, with any arguments being treated the same as command-line arguments, e.g.:

```
Prelude> let 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` command:

```
Prelude> let foo = putStrLn "foo" >> System.Environment.getArgs >>= print
Prelude> let 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"]
```

2.4.7 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> Time.getClockTime
Wed Mar 14 12:23:13 GMT 2001
Prelude> print it
Wed Mar 14 12:23:13 GMT 2001
```

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.

2.4.8 Type defaulting in GHCi

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 `-XExtendedDefaultRules` flag is given, the following additional differences apply:

- Rule 2 above is relaxed thus: *All* of the classes `Ci` are single-parameter type classes.
- Rule 3 above is relaxed this: At least one of the classes `Ci` is numeric, *or is `Show`, `Eq`, or `Ord`*.
- The unit type `()` is added to the start of the standard list of types which are tried when doing type defaulting.

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 Section 2.4.1 for how the monad of a computational expression defaults to `IO` if possible.

2.4.9 Using a custom interactive printing function

[**New in version 7.6.1**] By default, 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` 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.

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.sprinter SpecPrinter
```

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

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

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

The `-interactive-print` flag can also be used when running GHC in `-e mode`:

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

2.5 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 Section 2.5.6).

2.5.1 Breakpoints and inspecting variables

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

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

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

First, load the module into GHCi:

```
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>
```

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

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` 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` 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`, 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 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`, 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` does not force any evaluation. The idea is that `:print` 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 Section 2.5.3). Rather than forcing thunks, `:print` binds each thunk to a fresh variable beginning with an underscore, in this case `_t1`.

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

[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.

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

Now, since `:force` 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`. 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` 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.

2.5.1.1 Setting breakpoints

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

```
:break identifier
```

Where *identifier* names any top-level function in an interpreted module currently loaded into GHCi (qualified names may be used). The breakpoint will be set on the body of the function, when it is fully applied 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 `-fwarn-tabs` in Section 4.8).

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.

2.5.1.2 Listing and deleting breakpoints

The list of breakpoints currently enabled can be displayed using `:show breaks`:

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

To delete a breakpoint, use the `:delete` command with the number given in the output from `:show breaks`:

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

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

2.5.2 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` to enable all the breakpoints in the program, and execute until the next breakpoint is reached. Use `:steplocal` to limit the set of enabled breakpoints to those in the current top level function. Similarly, use `:stepmodule` 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`* begins the evaluation of *`expr`* in single-stepping mode. If *`expr`* is omitted, then it single-steps from the current breakpoint. `:steplocal` and `:stepmodule` work similarly.

The `:list` 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`:

```
[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>
```

2.5.3 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`:

```
... [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`:

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

2.5.4 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`). So it will probably be necessary to issue a `:continue` immediately when evaluating `_result`. Alternatively, you can use `:force` which ignores breakpoints.

2.5.5 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 (Chapter 5), 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` 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`:

```
[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` can be used to traverse forward in the history.

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

The history is only available when using `:trace`; the reason for this is we found that logging each breakpoint in the history cuts performance by a factor of 2 or more. By default, GHCi remembers the last 50 steps in the history, but this can be changed with the `-fghci-hist-size=`*n* option).

2.5.6 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 Section 5.3).

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` and `:history` 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` which causes the evaluator to stop when an exception is thrown, and `-fbreak-on-error`, 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`, 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.

2.5.7 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 (Section 2.5.1), 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.

2.5.8 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 Section 7.13.4) are only available at the scope of a breakpoint if there is an explicit type signature.

2.6 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 Section 2.7). 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 Chapter 4) also make sense in interactive mode. The ones that don't make sense are mostly obvious.

2.6.1 Packages

Most packages (see Section 4.9.1) 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` flag:

```
$ ghci -package readline
GHCi, version 6.8.1: 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`.

2.6.2 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 Section 2.2.1.) For example, to load the "m" library:

```
$ ghci -lm
```

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

- Paths specified using the `-Lpath` 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 Section 4.12.6).

2.7 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.

`:back` Travel back one step in the history. See Section 2.5.5. See also: `:trace`, `:history`, `:forward`.

`:break [`*`identifier`*` | [`*`module`*`] `*`line`*` [`*`column`*`]]` Set a breakpoint on the specified function or line and column. See Section 2.5.1.1.

`: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 (Section 2.4.5).

There are two variants of the browse command:

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

  The `*`-form is only available for modules which are interpreted; for compiled modules (including modules from packages) only the non-`*` form of `:browse` 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.listToMaybe :: [a] -> Maybe a
Data.Maybe.mapMaybe :: (a -> Maybe b) -> [a] -> [b]
Data.Maybe.maybeToList :: Maybe a -> [a]
-- imported via Prelude
Just :: a -> Maybe a
data Maybe a = Nothing | Just a
Nothing :: Maybe a
maybe :: b -> (a -> b) -> Maybe a -> b
```

  This output shows that, in the context of the current session (ie in the scope of `Prelude`), the first group of items from `Data.Maybe` are not in scope (althought they are available in fully qualified form in the GHCi session - see Section 2.4.5), 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` command for showing the current working directory.

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

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

`: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` prints the n^{th} to m^{th} completion candidates for the partial input *`string-literal`* for the completion domain denoted by *`type`*. Currently, only the `repl` domain is supported which denotes the kind of completion that would be provided interactively by GHCi at the input prompt.

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

The output of `:complete` 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`*] `:etags` [*`filename`*] Generates a "tags" file for Vi-style editors (`:ctags`) or Emacs-style editors (`:etags`). 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` 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 & time:

```
Prelude> let date _ = Time.getClockTime >>= print >> return ""
Prelude> :def date date
Prelude> :date
Fri Mar 23 15:16:40 GMT 2001
```

Here's an example of a command that takes an argument. It's a re-implementation of `:cd`:

```
Prelude> let mycd d = 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` 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`.

`:etags` See `:ctags`.

`:force identifier ...` Prints the value of *identifier* in the same way as `:print`. Unlike `:print`, `:force` 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` Move forward in the history. See Section 2.5.5. See also: `:trace`, `:history`, `:back`.

`: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`; see Section 2.5.5. To set the number of history entries stored by GHCi, use `-fghci-hist-size=n`.

`: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` or `:module` 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` 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` 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.

After a `:load` 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`, or
- `Prelude` otherwise.

:main ***arg_1 ...arg_n*** 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` command. This runs whatever `main` is in scope, with any arguments being treated the same as command-line arguments, e.g.:

```
Prelude> let 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` command:

```
Prelude> let foo = putStrLn "foo" >> System.Environment.getArgs >>= print
Prelude> let 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 [+|-] [*]*mod$_1$* ...[*]*mod$_n$*, import *mod* Sets or modifies the current context for statements typed at the prompt. The form `import mod` is equivalent to `:module +mod`. See Section 2.4.5 for more details.

:print *names* ... Prints a value without forcing its evaluation. `:print` 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` 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` binds a fresh variable with a name beginning with `_t` to each thunk. See Section 2.5.1 for more information. See also the `:sprint` command, which works like `:print` 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`) 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.

:run See `:main`.

: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`

:set [*option*...] Sets various options. See Section 2.8 for a list of available options and Section 4.19.10 for a list of GHCi-specific flags. The `:set` 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` 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 sequence `%s` is replaced by the names of the modules currently in scope, `%l` is replaced by the line number (as referenced in compiler messages) of the current prompt, and `%%` is replaced by `%`. If *prompt* starts with " then it is parsed as a Haskell String; otherwise it is treated as a literal string.

:set prompt2 *prompt* Sets the string to be used as the continuation prompt (used when using the `:{` command) 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` 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` 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` and `:cmd` you can use `:set stop` to implement conditional breakpoints:

```
*Main> :def cond \expr -> return (":cmd if (" ++ expr ++ ") then return \"\" else ↩
    return \":continue\"")
*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`, but options set with `:seti` affect only expressions and commands typed at the prompt, and not modules loaded with `:load` (in contrast, options set with `:set` apply everywhere). See Section 2.8.3.

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` 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` 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`).

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

:sprint Prints a value without forcing its evaluation. `:sprint` is similar to `:print`, 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 Section 2.5.2.

: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`. See Section 2.5.5.

:type *expression* Infers and prints the type of *expression*, including explicit forall quantifiers for polymorphic types. The monomorphism restriction is *not* applied to the expression during type inference.

:undef *name* Undefines the user-defined command *name* (see `:def` above).

:unset *option*... Unsets certain options. See Section 2.8 for a list of available options.

:! *command*... Executes the shell command *command*.

2.8 The :set and :seti commands

The `:set` command sets two types of options: GHCi options, which begin with '+', and "command-line" options, which begin with '-'.

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

2.8.1 GHCi options

GHCi options may be set using `:set` and unset using `:unset`.

The available GHCi options are:

+m Enable parsing of multiline commands. A multiline command is prompted for when the current input line contains open layout contexts (see Section 2.4.3).

+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.

+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.

+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`'.

2.8.2 Setting GHC command-line options in GHCi

Normal GHC command-line options may also be set using `:set`. For example, to turn on `-fwarn-missing-signatures`, you would say:

```
Prelude> :set -fwarn-missing-signatures
```

Any GHC command-line option that is designated as *dynamic* (see the table in Section 4.19), may be set using `:set`. To unset an option, you can set the reverse option:

```
Prelude> :set -fno-warn-incomplete-patterns -XNoMultiParamTypeClasses
```

Section 4.19 lists the reverse for each option where applicable.

Certain static options (`-package`, `-I`, `-i`, and `-l` in particular) will also work, but some may not take effect until the next reload.

2.8.3 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` command modifies both, but there is also a `:seti` 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 `-XMonoLocalBinds` 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` rather than `:set`, 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` and `:seti` 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 Section 2.9. 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 Section 2.4.8).
- The Monomorphism Restriction is disabled (see Section 7.13.9.1).

2.9 The `.ghci` file

When it starts, unless the `-ignore-dot-ghci` 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 (eg. `:set +s`), and defining useful macros. Note: when setting language options in this file it is usually desirable to use `:seti` rather than `:set` (see Section 2.8.3).

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` flag is a static one, but in fact it works to set it using `:set` like this. The changes won't take effect until the next `:load`, 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` flags will be read after the standard files, allowing the use of custom .ghci files.

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

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

`-ghci-script` Read a specific file after the usual startup files. 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.

2.10 Compiling to object code inside GHCi

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

2.11 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` 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` 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` switch, which is the default. Consult whoever supplied your GHCi installation.

After using `getContents`, I can't use `stdin` again 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` or `:reload` 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 Section 12.2.

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

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

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

Chapter 3

Using runghc

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

3.1 Flags

The runghc commandline looks like:

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

The runghc flags are `-f /path/to/ghc`, which tells runghc which GHC to use to run the program, and `--help`, which prints usage information. If it is not given then runghc will search for GHC in the directories in the system search path.

runghc will try to work out where the boundaries between `[runghc flags]` and `[GHC flags]`, and `[program args]` and `module` are, but you can use a `--` flag if it doesn't get it right. For example, `runghc ---fwarn-unused-bindings Foo` means runghc won't try to use `warn-unused-bindings` as the path to GHC, but instead will pass the flag to GHC. If a GHC flag doesn't start with a dash then you need to prefix it with `--ghc-arg=` or runghc will think that it is the program to run, e.g. `runghc -package-db --ghc-arg=foo.conf Main.hs`.

Chapter 4

Using GHC

4.1 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` 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.

4.2 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:

4.2.1 Command-line arguments

An invocation of GHC takes the following form:

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

Command-line arguments are either options or file names.

Command-line options begin with `-`. They may *not* be grouped: `-vO` is different from `-v -O`. 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`.

4.2.2 Command line options in source files

Sometimes it is useful to make the connection between a source file and the command-line options it requires quite tight. For instance, if a Haskell source file deliberately uses name shadowing, it should be compiled with the `-fno-warn-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 :

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

`OPTIONS_GHC` is a *file-header pragma* (see Section 7.22).

Only *dynamic* flags can be used in an `OPTIONS_GHC` pragma (see Section 4.3).

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

4.2.3 Setting options in GHCi

Options may also be modified from within GHCi, using the `:set` command. See Section 2.8 for more details.

4.3 Static, Dynamic, and Mode options

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

Mode flags For example, `--make` or `-E`. There may only be a single mode flag on the command line. The available modes are listed in Section 4.5.

Dynamic Flags 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` in GHCi.

Static Flags 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 (Section 4.19) 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` command; these are listed as "static/`:set`" in the table.

4.4 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.

4.5 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.

If no mode flag is present, then GHC will enter make mode (Section 4.5.1) 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:

`ghc --interactive` Interactive mode, which is also available as **ghci**. Interactive mode is described in more detail in Chapter 2.

`ghc --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 Section 4.5.1.

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

`ghc -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 Section 4.5.2 for more details.

`ghc -E` `ghc -C` `ghc -S` `ghc -c` 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 Section 4.5.3.

`ghc -M` Dependency-generation mode. In this mode, GHC can be used to generate dependency information suitable for use in a `Makefile`. See Section 4.7.12.

`ghc --mk-dll` DLL-creation mode (Windows only). See Section 12.6.1.

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

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

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

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

ghc --info Print information about the compiler.

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

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

ghc --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 Section 4.9).

4.5.1 Using ghc --make

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

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

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

```
ghc Main.hs
```

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

The main advantages to using `ghc --make` over traditional `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` flag, you can compile modules in parallel. Specify `-jN` to compile *N* jobs in parallel.

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 Section 4.2.2).

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`, 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` option can be used to add directories to the search path (see Section 4.7.3).

4.5.2 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
```

4.5.3 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 Section 4.11 for more details.

Note: C pre-processing is optional, the -cpp flag turns it on. See Section 4.12.3 for more details.

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

Note: The option -C is only available when GHC is built in unregisterised mode. See Section 4.11.4 for more details.

4.5.3.1 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 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.

4.6 Verbosity options

See also the `--help`, `--version`, `--numeric-version`, and `--print-libdir` modes in Section 4.5.

-v The `-v` 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` or `--interactive` is on).

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

-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-explicit-foralls, -fprint-explicit-kinds These two flags control the way in which GHC displays types, in error messages and in GHCi. Using `-fprint-explicit-foralls` 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
```

Using `-fprint-explicit-kinds` 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
```

-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 Section 4.17.3.

-Rghc-timing Prints a one-line summary of timing statistics for the GHC run. This option is equivalent to `+RTS -tstderr`, see Section 4.17.3.

4.7 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.

Note that this section is written with *hierarchical modules* in mind (see Section 7.3.6); hierarchical modules are an extension to Haskell 98 which extends the lexical syntax of module names to include a dot '.'. Non-hierarchical modules are thus a special case in which none of the module names contain dots.

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.

4.7.1 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 encodings 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.

4.7.2 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` option instead (see Section 4.7.7).

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` and `-hisuf` instead of `-odir` and `-osuf` 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 Section 4.7.3. 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` option, and the name of the interface file can be specified directly using the `-ohi` option.

4.7.3 The search path

In your program, you import a module `Foo` by saying `import Foo`. In `--make` mode or GHCi, GHC will look for a source file for `Foo` and arrange to compile it first. Without `--make`, 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` mode or GHCi, or *`hisuf`* otherwise.

For example, suppose the search path contains directories `d1`, `d2`, and `d3`, and we are in `--make` 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`*`dirs`* 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 (Section 4.9) for details.

4.7.4 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/Wurble.iface` with the option `-ohi bar2/Wurble.iface` (not recommended).

WARNING: if you redirect the interface file somewhere that GHC can't find it, then the recompilation checker may get confused (at the least, you won't get any recompilation avoidance). We recommend using a combination of `-hidir` and `-hisuf` options instead, if possible.

To avoid generating an interface at all, you could use this option to redirect the interface into the bit bucket: `-ohi /dev/null`, for example.

-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 Section 8.2.1). 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`, `-hidir`, `-stubdir` and `-dumpdir`.

-osuf *suffix*, -hisuf *suffix*, -hcsuf *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.

Similarly, the `-hisuf suffix` will change the `.hi` file suffix for non-system interface files (see Section 4.7.7).

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

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 -auto-all
```

to get the profiled version.

4.7.5 Keeping Intermediate Files

The following options are useful for 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 (NOTE: `.hc` files are only generated by unregisterised compilers).

-keep-llvm-file, -keep-llvm-files Keep intermediate `.ll` files when doing `.hs`-to-`.o` compilations via LLVM (NOTE: `.ll` files aren't generated when using the native code generator, you may need to use `-fllvm` to force them to be produced).

-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 Section 4.7.6). Running GHC with `-v` will show you what temporary files were generated along the way.

4.7.6 Redirecting temporary files

-tmpdir 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>` 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.

Even better idea: Set the `DEFAULT_TMPDIR` make variable when building GHC, and never worry about `TMPDIR` again. (see the build documentation).

4.7.7 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` 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` 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 (ish) format. See Section 4.5.

4.7.8 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.

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.

4.7.9 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 an 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 Section 4.7.11).
  - 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 Section 7.4.1), 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 (Section 7.13.5), 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 class and datatype parameters is now representational. To get another role, use a role annotation. (See Section 7.27.)

4.7.10 Module signatures

GHC supports the specification of module signatures, which both implementations and users can typecheck against separately. This functionality should be considered experimental for now; some details, especially for type classes and type families, may change. This system was originally described in Backpack: Retrofitting Haskell with Interfaces. Signature files are somewhat similar to `hs-boot` files, but have the `hsig` extension and behave slightly differently.

Suppose that I have modules `String.hs` and `A.hs`, thus:

```
module Text where
    data Text = Text String

    empty :: Text
    empty = Text ""

    toString :: Text -> String
    toString (Text s) = s

module A where
    import Text
    z = toString empty
```

Presently, module `A` depends explicitly on a concrete implementation of `Text`. What if we wanted to a signature `Text`, so we could vary the implementation with other possibilities (e.g. packed UTF-8 encoded bytestrings)? To do this, we can write a signature `TextSig.hsig`, and modify `A` to include the signature instead:

```
module TextSig where
    data Text
    empty :: Text
    toString :: Text -> String

module A where
    import TextSig
    z = toString empty
```

To compile these two files, we need to specify what module we would like to use to implement the signature. This can be done by compiling the implementation, and then using the `-sig-of` flag to specify the implementation backing a signature:

```
ghc -c Text.hs
ghc -c TextSig.hsig -sig-of main:Text
ghc -c A.hs
```

Signature files can also be compiled as part of `--make`, in which case the syntax is extended to support specifying implementations of multiple signatures as `FooSig is main:Foo, BarSig is main:Bar`. At the moment, you must specify the full module name (package key, colon, and then module name), although in the future we may support more user-friendly syntax.

To just type-check an interface file, no `-sig-of` is necessary; instead, just pass the options `-fno-code -fwrite-interface`. `hsig` files will generate normal interface files which other files can also use to type-check against. However, at the moment, we always assume that an entity defined in a signature is a unique identifier (even though we may happen to know it is type equal with another identifier). In the future, we will support passing shaping information to the compiler in order to let it know about these type equalities.

Just like `hs-boot` files, when an `hsig` file is compiled it is checked for type consistency against the backing implementation. Signature files are also written in a subset of Haskell essentially identical to that of `hs-boot` files.

There is one important gotcha with the current implementation: currently, instances from backing implementations will "leak" code that uses signatures, and explicit instance declarations in signatures are forbidden. This behavior will be subject to change.

4.7.11 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
OBJS = Main.o   Foo.o   Bar.o

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

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

# Standard suffix rules
.o.hi:
        @:

.lhs.o:
        $(HC) -c $< $(HC_OPTS)

.hs.o:
        $(HC) -c $< $(HC_OPTS)

.o-boot.hi-boot:
```

```
        @:

.lhs-boot.o-boot:
        $(HC) -c $< $(HC_OPTS)

.hs-boot.o-boot:
        $(HC) -c $< $(HC_OPTS)

# Inter-module dependencies
Foo.o Foo.hc Foo.s    : Baz.hi          # Foo imports Baz
Main.o Main.hc Main.s : Foo.hi Baz.hi   # Main imports Foo and Baz
```

(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 Section 4.7.9).

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 Section 4.7.12

4.7.12 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 -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 Section 4.7.9 for details of `hi-boot` style interface files.)

If `M` imports multiple modules, then there will be multiple lines with `M.o` as the target.

There is no need to list all of the source files as arguments to the **ghc -M** command; **ghc** traces the dependencies, just like **ghc --make** (a new feature in GHC 6.4).

Note that `ghc -M` needs to find a *source file* for each module in the dependency graph, so that it can parse the import declarations and follow dependencies. Any pre-compiled modules without source files must therefore belong to a package[1].

By default, **ghc -M** generates all the dependencies, and then concatenates them onto the end of `makefile` (or `Makefile` if `makefile` doesn't exist) bracketed by the lines "`#DO NOT DELETE:Beginning of Haskell dependencies`" and "`#DO NOT DELETE:End of Haskell dependencies`". If these lines already exist in the `makefile`, then the old dependencies are deleted first.

Don't forget to use the same `-package` options on the `ghc -M` command line as you would when compiling; this enables the dependency generator to locate any imported modules that come from packages. The package modules won't be included in the dependencies generated, though (but see the `--include-pkg-deps` option below).

The dependency generation phase of GHC can take some additional options, which you may find useful. The options which affect dependency generation are:

`-ddump-mod-cycles` Display a list of the cycles in the module graph. This is useful when trying to eliminate such cycles.

`-v2` Print a full list of the module dependencies to stdout. (This is the standard verbosity flag, so the list will also be displayed with `-v3` and `-v4`; Section 4.6.)

`-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 <suf>` Make extra 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`. (Useful in conjunction with NoFib "ways".)

`--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.

4.7.13 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 Section 7.6.3.3), 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:

[1] This is a change in behaviour relative to 6.2 and earlier.

```
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* contains at least one *orphan instance* or at least one *orphan rule*.
- An instance declaration in a module M is an *orphan instance* if
  - The class of the instance declaration is not declared in M, and
  - *Either* the class has no functional dependencies, and none of the type constructors in the instance head is declared in M; *or* there is a functional dependency for which none of the type constructors mentioned in the *non-determined* part of the instance head is defined in M.

  Only the instance head counts. In the example above, it is not good enough for C's declaration to be in module A; it must be the declaration of D or T.
- A rewrite rule in a module M is an *orphan rule* 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 `-fwarn-orphans`, GHC will warn you if you are creating an orphan module. Like any warning, you can switch the warning off with `-fno-warn-orphans`, and `-Werror` 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 mode. If there is a `[orphan module]` on the first line, GHC considers it an orphan module.

4.8 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: `-fwarn-overlapping-patterns`, `-fwarn-warnings-deprecations`, `-fwarn-deprecated-flags`, `-fwarn-unrecognised-pragmas`, `-fwarn-pointless-pragmas`, `-fwarn-duplicate-constraints`, `-fwarn-duplicate-exports`, `-fwarn-overflowed-literals`, `-fwarn-empty-enumerations`, `-fwarn-missing-fields`, `-fwarn-missing-methods`, `-fwarn-wrong-do-bind`, `-fwarn-unsupported-calling-conventions`, `-fwarn-dodgy-foreign-imports`, `-fwarn-inline-rule-shadowing`, `-fwarn-unsupported-llvm-version`, `-fwarn-context-quantification`, and `-fwarn-tabs`. The following flags are simple ways to select standard “packages” of warnings:

`-W`: Provides the standard warnings plus `-fwarn-unused-binds`, `-fwarn-unused-matches`, `-fwarn-unused-imports`, `-fwarn-incomplete-patterns`, `-fwarn-dodgy-exports`, and `-fwarn-dodgy-imports`.

-Wall: Turns on all warning options that indicate potentially suspicious code. The warnings that are *not* enabled by `-Wall` are `-fwarn-type-defaults`, `-fwarn-name-shadowing`, `-fwarn-missing-signatures`, `-fwarn-warn-hi-shadowing`, `-fwarn-orphans`, `-fwarn-unused-do-bind`, and `-fwarn-trustworthy-safe`, `-fwarn-unticked-promoted-constructors`.

-w: Turns off all warnings, including the standard ones and those that `-Wall` doesn't enable.

-Werror: Makes any warning into a fatal error. Useful so that you don't miss warnings when doing batch compilation.

-Wwarn: Warnings are treated only as warnings, not as errors. This is the default, but can be useful to negate a `-Werror` flag.

The full set of warning options is described below. To turn off any warning, simply give the corresponding `-fno-warn-...` option on the command line.

-fwarn-typed-holes: Determines whether the compiler reports typed holes warnings. Has no effect unless typed holes errors are deferred until runtime. See Section 7.14 and Section 7.16

This warning is on by default.

-fdefer-type-errors: 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 Section 7.16

-fdefer-typed-holes: Defer typed holes errors until runtime. This will turn the errors produced by typed holes into warnings. Using a value that depends on a typed hole produces a runtime error, the same as `-fdefer-type-errors` (which implies this option). See Section 7.14 and Section 7.16.

Implied by `-fdefer-type-errors`. See also `-fwarn-typed-holes`.

-fwarn-partial-type-signatures: Determines whether the compiler reports holes in partial type signatures as warnings. Has no effect unless `-XPartialTypeSignatures` is enabled, which controls whether errors should be generated for holes in types or not. See Section 7.15.

This warning is on by default.

-fhelpful-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.

-fwarn-unrecognised-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.

-fwarn-pointless-pragmas: Causes a warning to be emitted when GHC detects that a module contains a pragma that has no effect.

This option is on by default.

-fwarn-warnings-deprecations: Causes a warning to be emitted when a module, function or type with a WARNING or DEPRECATED pragma is used. See Section 7.22.4 for more details on the pragmas.

This option is on by default.

-fwarn-amp: Causes a warning to be emitted when a definition is in conflict with the AMP (Applicative-Monad proosal), namely: 1. Instance of Monad without Applicative; 2. Instance of MonadPlus without Alternative; 3. Custom definitions of join/pure/<*>

This option is on by default.

-fwarn-deprecated-flags: Causes a warning to be emitted when a deprecated commandline flag is used.

This option is on by default.

-fwarn-unsupported-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`.

-fwarn-dodgy-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.

-fwarn-dodgy-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.

-fwarn-dodgy-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.

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

-fwarn-overflowed-literals-fwarn-empty-enumerations: Causes a warning to be emitted if an enumeration is empty, e.g. `[5 ..3]`.

-fwarn-lazy-unlifted-bindings: This flag is a no-op, and will be removed in GHC 7.10.

-fwarn-duplicate-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 on by default.

-fwarn-duplicate-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.

-fwarn-hi-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.

-fwarn-identities: 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`.

-fwarn-implicit-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` or a `LANGUAGE NoImplicitPrelude` pragma).

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

-fwarn-incomplete-patterns, -fwarn-incomplete-uni-patterns: The option `-fwarn-incomplete-patterns` 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 `-fwarn-incomplete-patterns` 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`.

The flag `-fwarn-incomplete-uni-patterns` is similar, except that it applies only to lambda-expressions and pattern bindings, constructs that only allow a single pattern:

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

-fwarn-incomplete-record-updates: The function `f` below will fail when applied to `Bar`, so the compiler will emit a warning about this when `-fwarn-incomplete-record-updates` 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.

-fwarn-missing-fields: This option is on by default, and warns you whenever the construction of a labelled field constructor isn't complete, missing initializers 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.

-fwarn-missing-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 `-fwarn-import-lists` 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`.

-fwarn-missing-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 Section 7.22.5.

-fwarn-missing-signatures: If you would like GHC to check that every top-level function/value has a type signature, use the `-fwarn-missing-signatures` option. As part of the warning GHC also reports the inferred type. The option is off by default.

-fwarn-missing-exported-sigs: 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 `-fwarn-missing-exported-sigs` option. This option takes precedence over `-fwarn-missing-signatures`. As part of the warning GHC also reports the inferred type. The option is off by default.

-fwarn-missing-local-sigs: If you use the `-fwarn-missing-local-sigs` 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.

-fwarn-name-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) }
```

-fwarn-orphans, -fwarn-auto-orphans: 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 Section 4.7.13 for details.

The flag `-fwarn-orphans` warns about user-written orphan rules or instances. The flag `-fwarn-auto-orphans` warns about automatically-generated orphan rules, notably as a result of specialising functions, for type classes (`Specialise`) or argument values (`-fspec-constr`).

-fwarn-overlapping-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.

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

This warning is off by default.

-fwarn-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.

-fwarn-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.

-fwarn-unticked-promoted-constructors: Warn if a promoted data constructor is used without a tick preceding it's 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 off by default.

`-fwarn-unused-binds`: Report any function definitions (and local bindings) 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  -- Warning about unused q
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
```

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

```
Just _ = rhs3    -- Warning: unused pattern binding
(_, _) = rhs4    -- Warning: unused pattern binding
_  = rhs3        -- No warning: lone wild-card pattern
!_ = rhs4        -- No warning: banged wild-card pattern; behaves like seq
```

  The motivation for allowing lone wild-card patterns is they are not very different from `_v =rhs3`, which elicits no warning; and they can be useful to add a type constraint, e.g. `_ =x::Int`. A lone banged wild-card pattern is is useful as an alternative (to `seq`) way to force evaluation.

`-fwarn-unused-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.

`-fwarn-unused-matches`: Report all unused variables which arise from 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
```

`-fwarn-unused-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 }
```

`-fwarn-context-quantification`: Report if a variable is quantified only due to its presence in a context (see Section 7.13.6). For example,

```
type T a = Monad m => a -> f a
```

It is recommended to write this polymorphic type as

```
type T a = forall m. Monad m => a -> f a
```

instead.

-fwarn-wrong-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 }
```

-fwarn-inline-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 Section 7.23.3.

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

4.9 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 Section 4.9.1.

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).

4.9.1 Using Packages

GHC only knows about packages that are *installed*. To see which packages are installed, 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 (eg. `(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. If there are a 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 Section 7.29) with the `-fpackage-trust` flag enabled.

To see which modules are provided by a package use the `ghc-pkg` command (see Section 4.9.7):

```
$ 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 P` This option causes the installed package *P* to be exposed. The package *P* 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 *P* installed and `-hide-all-packages` was not specified, then all other versions will become hidden. `-package` supports thinning and renaming described in Section 4.9.4.

The `-package P` option also causes package *P* 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`/`-dynamic`.

In `--make` mode and `--interactive` mode (see Section 4.5), the compiler normally determines which packages are required by the current Haskell modules, and links only those. In batch mode however, the dependency information isn't available, and explicit `-package` options must be given when linking. The one other time you might need to use `-package` to force linking a package is when the package does not contain any Haskell modules (it might contain a C library only, for example). In that case, GHC will never discover a dependency on it, so it has to be mentioned explicitly.

For example, to link a program consisting of objects `Foo.o` and `Main.o`, where we made use of the `network` package, we need to give GHC the `-package` flag thus:

```
$ ghc -o myprog Foo.o Main.o -package network
```

The same flag is necessary even if we compiled the modules from source, because GHC still reckons it's in batch mode:

```
$ ghc -o myprog Foo.hs Main.hs -package network
```

`-package-id P` Exposes a package like `-package`, 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 `-package-id` flags to GHC. `-package-id` supports thinning and renaming described in Section 4.9.4.

`-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` 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 P` This option does the opposite of `-package`: 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 P` Causes the compiler to behave as if package *P*, and any packages that depend on `P`, are not installed at all.

Saying `-ignore-package P` is the same as giving `-hide-package` flags for `P` and all the packages that depend on `P`. Sometimes we don't know ahead of time which packages will be installed that depend on `P`, which is when the `-ignore-package` 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-package-key foo` Tells GHC the the module being compiled forms part of package key *foo*; internally, these keys are used to determine type equality and linker symbols. If this flag is omitted (a very common case) then the default package `main` is assumed.

`-trust P` This option causes the install package *P* to be both exposed and trusted by GHC. This command functions in the in a very similar way to the `-package` command but in addition sets the selected packaged to be trusted by GHC, regardless of the contents of the package database. (see Section 7.29).

`-distrust P` This option causes the install package *P* to be both exposed and distrusted by GHC. This command functions in the in a very similar way to the `-package` command but in addition sets the selected packaged to be distrusted by GHC, regardless of the contents of the package database. (see Section 7.29).

`-distrust-all` 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` options. This option does not change the exposed/hidden status of a package, so it isn't equivalent to applying `-distrust` to all packages on the system. (see Section 7.29).

4.9.2 The main package

Every complete Haskell program must define `main` in module `Main` in package `main`. (Omitting the `-this-package-key` 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
```

4.9.3 Consequences of packages for the Haskell language

It is possible that by using packages you might end up with a program that contains two modules with the same name: perhaps you used a package P that has a *hidden* module M, and there is also a module M in your program. Or perhaps the dependencies of packages that you used contain some overlapping modules. Perhaps the program even contains multiple versions of a certain package, due to dependencies from other packages.

None of these scenarios gives rise to an error on its own[2], 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.

4.9.4 Thinning and renaming modules

When incorporating packages from multiple sources, you may end up in a situation where multiple packages publish modules with the same name. Previously, the only way to distinguish between these modules was to use [?title]. However, since GHC 7.10, the `-package` 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 Section 7.3.16 extension.

4.9.5 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 Section 4.9.7).

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`.
- A 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 Section 4.9.7).

[2] it used to in GHC 6.4, but not since 6.6

When GHC starts up, it reads the contents of these two package databases, and builds up a list of the packages it knows about. You can see GHC's package table by running GHC with the `-v` flag.

Package databases may overlap, and they are arranged in a stack structure. Packages closer to the top of the stack will override (*shadow*) those below them. By default, the stack contains just the global and the user's package databases, in that order.

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. Packages in additional databases read this way will override those in the initial stack and those in previously specified databases.

-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` 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` 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` to specify the position in the stack where the user's package database should be loaded.

4.9.5.1 The GHC_PACKAGE_PATH environment variable

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 is used by GHC and ghc-pkg, with earlier databases in the list overriding later ones. This order was chosen to match the behaviour of the `PATH` environment variable; think of it as a list of package databases that are searched left-to-right for packages.

If `GHC_PACKAGE_PATH` ends in a separator, then the default package database stack (i.e. the user and global package databases, in that order) is appended. 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:
```

(use `;` instead of `:` on Windows).

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.

4.9.6 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.

4.9.7 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` environment variable (see Section 4.9.5.1, and using `--package-db` 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` 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 Section 4.9.9.

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 Section 4.9.9 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).

4.9.8 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 Section 4.9.9.

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` then building the `HSfoo.o` is worthwhile because it saves time when loading the package into GHCi. Without `-split-objs`, 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`/`-dynamic` flags work, see Section 4.12.6.

  ```
  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 Section 4.12.5 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 `-this-package-key` option (Section 4.9.1). Failure to use the `-this-package-key` option 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.

4.9.9 `InstalledPackageInfo`: a package specification

A package specification is a Haskell record; in particular, it is the record InstalledPackageInfo in the module Distribution.InstalledPackag which is part of the Cabal package distributed with GHC.

An `InstalledPackageInfo` has a human readable/writable syntax. The functions `parseInstalledPackageInfo` and `showInstalledPackageInfo` read and write this syntax respectively. Here's an example of the `InstalledPackageInfo` for the `unix` package:

```
$ ghc-pkg describe unix
name: unix
version: 2.3.1.0
id: unix-2.3.1.0-de7803f1a8cd88d2161b29b083c94240
license: BSD3
copyright:
maintainer: libraries@haskell.org
stability:
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` The package's name (without the version).

`id` The installed package ID. It is up to you to choose a suitable one.

`version` 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 `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 (eg. stable, provisional or experimental).

`homepage` (optional freeform) URL of the package's home page.

`package-url` (optional freeform) URL of a downloadable distribution for this package. The distribution should be a Cabal package.

`description` (optional freeform) Description of the package.

`category` (optional freeform) Which category the package belongs to. This field is for use in conjunction with a future centralised package distribution framework, tentatively titled Hackage.

`author` (optional freeform) Author of the package.

`exposed` (bool) Whether the package is exposed or not.

`exposed-modules` (string list) modules exposed by this package.

`hidden-modules` (string list) modules provided by this package, but not exposed to the programmer. These modules cannot be imported, but they are still subject to the overlapping constraint: no other package in the same program may provide a module of the same name.

`reexported-modules` Modules reexported by this package. This list takes the form of `pkg:OldName as NewName (A@orig-pkg-0.1-HASH)`: the first portion of the string is the user-written reexport specification (possibly omitting the package qualifier and the renaming), while the parenthetical is the original package which exposed the module under are particular name. Reexported modules have a relaxed overlap constraint: it's permissible for two packages to reexport the same module as the same name if the reexported moduleis identical.

trusted (bool) Whether the package is trusted or not.

import-dirs (string list) A list of directories containing interface files (`.hi` files) for this package.

If the package contains profiling libraries, then the interface files for those library modules should have the suffix `.p_hi`. So the package can contain both normal and profiling versions of the same library without conflict (see also `library_dirs` below).

library-dirs (string list) A list of directories containing libraries for this package.

hs-libraries (string list) A list of libraries containing Haskell code for this package, with the `.a` or `.dll` suffix omitted. When packages are built as libraries, the `lib` prefix is also omitted.

For use with GHCi, each library should have an object file too. The name of the object file does *not* have a `lib` prefix, and has the normal object suffix for your platform.

For example, if we specify a Haskell library as `HSfoo` in the package spec, then the various flavours of library that GHC actually uses will be called:

libHSfoo.a The name of the library on Unix and Windows (mingw) systems. 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.

4.10 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.

4.10.1 `-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 eg. **make** has inserted a `-O` on the command line already.

`-O` or `-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.

At the moment, `-O2` is *unlikely* to produce better code than `-O`.

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

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`, then stand back in amazement.

4.10.2 `-f*`: platform-independent flags

These flags turn on and off individual optimisations. Flags marked as *Enabled 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`. See Section 4.19.15 for a compact list.

`-fcase-merge` *On by default.* Merge immediately-nested case expressions that scrutinse the same variable. Example

```
  case x of
     Red -> e1
     _   -> case x of
              Blue -> e2
              Green -> e3
==>
  case x of
     Red -> e1
     Blue -> e2
     Green -> e2
```

`-fcall-arity` *On by default.*.

-fcmm-elim-common-blocks *On by default.*. Enables the common block elimination optimisation in the code generator. This optimisation attempts to find identical Cmm blocks and eliminate the duplicates.

-fcmm-sink *On by default.*. 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-off Switch off CPR analysis in the demand analyser.

-fcse *On by default.*. 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.

-fdicts-cheap A very experimental flag that makes dictionary-valued expressions seem cheap to the optimiser.

-fdicts-strict Make dictionaries strict.

-fdmd-tx-dict-sel *On by default for* `-O0`*,* `-O`*,* `-O2`*.*

Use a special demand transformer for dictionary selectors.

-fdo-eta-reduction *On by default.* Eta-reduce lambda expressions, if doing so gets rid of a whole group of lambdas.

-fdo-lambda-eta-expansion *On by default.* Eta-expand let-bindings to increase their arity.

-feager-blackholing 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.

-fexcess-precision 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 Section 13.2.1.

-fexpose-all-unfoldings 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 *On by default.* Float let-bindings inwards, nearer their binding site. See Let-floating: moving bindings to give faster pr

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 *On by default.* 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 faste 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 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. Off by default.

-fignore-asserts *On by default.*. Causes GHC to ignore uses of the function `Exception.assert` in source code (in other words, rewriting `Exception.assert p e` to `e` (see Section 7.20).

-fignore-interface-pragmas 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 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` 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 *Off by default, but enabled by -O2.* 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`) but for free variables rather than arguments.

-fliberate-case-threshold=*n* Set the size threshold for the liberate-case transformation. Default: 2000

-floopification *On by default.*

When this optimisation is enabled the code generator will turn all self-recursive saturated tail calls into local jumps rather than function calls.

-fmax-inline-alloc-size=*n* Set the maximum size of inline array allocations to n bytes (default: 128). GHC will allocate non-pinned arrays of statically known size in the current nursery block if they're no bigger than n bytes, ignoring GC overheap. This value should be quite a bit smaller than the block size (typically: 4096).

-fmax-inline-memcpy-insn=*n* Inline memcpy calls if they would generate no more than n pseudo instructions (default: 32).

-fmax-inline-memset-insns=*n* Inline memset calls if they would generate no more than n pseudo instructions (default: 32).

-fmax-relevant-binds=*n* 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. The default is 6. 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-simplifier-iterations=*n* Sets the maximal number of iterations for the simplifier. Defult: 4.

-fmax-worker-args=*n* If a worker has that many arguments, none will be unpacked anymore (default: 10)

-fno-opt-coercion Turn off the coercion optimiser.

-fno-pre-inlining Turn off pre-inlining.

-fno-state-hack Turn off the "state hack" whereby any lambda with a `State#` token as argument is considered to be single-entry, hence it is considered OK 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 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 *On by default.* 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 Make GHC be more precise about its treatment of bottom (but see also `-fno-state-hack`). 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 *Off by default due to a performance regression bug. Only applies in combination with the native code generator.* Use the graph colouring register allocator for register allocation in the native code generator. By default, GHC uses a simpler, faster linear register allocator. The downside being that the linear register allocator usually generates worse code.

-fregs-iterative *Off by default, 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` one but also enables iterative coalescing during register allocation.

-fsimplifier-phases=*n* Set the number of phases for the simplifier (default 2). Ignored with -O0.

-fsimpl-tick-factor=*n* GHC's optimiser can diverge if you write rewrite rules (Section 7.23) that don't terminate, or (less satisfactorily) if you code up recursion through data types (Section 13.2.1). 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 *Off by default, but enabled by -O2.* Turn on call-pattern specialisation; see Call-pattern specialisation for Haskell p

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 specialization very aggressively on `foldl_loop` due to the use of `SPEC` in the argument of the loop body. `SPEC` from `GHC.Types` is specifically recognized 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-count=*n* Set the maximum number of specialisations that will be created for any one function by the SpecConstr transformation (default: 3).

-fspec-constr-threshold=*n* Set the size threshold for the SpecConstr transformation (default: 2000).

-fspecialise *On by default.* Specialise each type-class-overloaded function defined in this module for the types at which it is called in this module. Also specialise imported functions that have an INLINABLE pragma (Section 7.22.6.2) for the types at which they are called in this module.

-fstatic-argument-transformation 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 *On by default.*. Switch on the strictness analyser. There is a very old paper about GHC's strictness analyser, Measuring the effectiveness of a simple strictness analyser, but the current one is quite a bit different.

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: *On by default.*. 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 Section 7.22.11) 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 Section 7.2) 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 Section 7.22.12).

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: 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 Section 7.22.11).

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 Section 7.22.12).

-funfolding-creation-threshold=*n*: (Default: 750) Governs the maximum size that GHC will allow a function unfolding to be. (An unfolding has a "size" that reflects the cost in terms of "code bloat" of expanding (aka inlining) that unfolding at a call site. A bigger function would be assigned a bigger cost.)

Consequences: (a) nothing larger than this will be inlined (unless it has an INLINE pragma); (b) nothing larger than this will be spewed into an interface file.

Increasing this figure is more likely to result in longer compile times than faster code. The `-funfolding-use-threshold` is more useful.

-funfolding-dict-discount=*n*: Default: 30

-funfolding-fun-discount=*n*: Default: 60

-funfolding-keeness-factor=*n*: Default: 1.5

`-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` 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` Part of Data Parallel Haskell (DPH).

On by default. 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 of unvectorised and if so, do just that.

`-fvectorise` Part of Data Parallel Haskell (DPH).

Off by default. 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.

4.11 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.

4.11.1 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.

4.11.2 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` flag. Currently *LLVM 2.8* and later are supported.

You must install and have LLVM available on your PATH for the LLVM code generator to work. Specifically GHC needs to be able to call the **opt** and **llc** tools. Secondly, if you are running Mac OS X with LLVM 3.0 or greater then you also need the Clang c compiler compiler available on your PATH.

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.

4.11.3 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 unregisterised mode, a mode where GHC produces 'portable' C code as output to facilitate porting GHC itself to a new platform. This mode produces much slower code though so it's unlikely your version of GHC was built this way. If it has then the native code generator probably won't be available. You can check this information by calling ghc --info.

4.11.4 Unregisterised compilation

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

When GHC is build in unregisterised 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 unregisterised mode.

Unregisterised compilation can be useful when porting GHC to a new machine, since it reduces the prerequisite tools to **gcc**, **as**, and **ld** and nothing more, and furthermore the amount of platform-specific code that needs to be written in order to get unregisterised compilation going is usually fairly small.

Unregisterised compilation cannot be selected at compile-time; you have to build GHC with the appropriate options set. Consult the GHC Building Guide for details.

You can check if your GHC is unregisterised by calling ghc --info.

4.12 Options related to a particular phase

4.12.1 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 Section 4.12.6.

-pgmlibtool *cmd* Use *cmd* as the libtool command (when using `-staticlib` only).

4.12.2 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 Section 4.12.4).

-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 Section 4.12.6.

So, for example, to force an `-Ewurble` option to the assembler, you would tell the driver `-opta-Ewurble` (the dash before the E is required).

GHC is itself a Haskell program, so if you need to pass options directly to GHC's runtime system you can enclose them in `+RTS ... -RTS` (see Section 4.17).

4.12.3 Options affecting the C pre-processor

-cpp The C pre-processor **cpp** is run over your Haskell code only if the `-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. NB: does *not* affect `-D` macros passed to the C compiler when compiling via C! For those, use the `-optc-Dfoo` hack... (see Section 4.12.2).

-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).

The symbols defined by GHC are listed below. To check which symbols are defined by your local GHC installation, the following trick is useful:

```
$ ghc -E -optP-dM -cpp foo.hs
$ cat foo.hspp
```

(you need a file `foo.hs`, but it isn't actually used).

__GLASGOW_HASKELL__ For version `x.y.z` of GHC, the value of `__GLASGOW_HASKELL__` is the integer `xyy` (if `y` is a single digit, then a leading zero is added, so for example in version 6.2 of GHC, `__GLASGOW_HASKELL__==602`). More information in Section 1.4.

With any luck, `__GLASGOW_HASKELL__` will be undefined in all other implementations that support C-style pre-processing.

(For reference: 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.

NB. 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
```

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_TH__ This is set to `YES` when the compiler supports Template Haskell, and to `NO` 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 *xy*.

__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.).

4.12.3.1 CPP and string gaps

A small word of warning: `-cpp` is not friendly to "string gaps".. In other words, strings such as the following:

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

don't work with `-cpp`; `/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 **cpp**s) leaves the backslash-space pairs alone and the string gap works as expected.

4.12.4 Options affecting a Haskell pre-processor

-F A custom pre-processor is run over your Haskell source file only if the `-F` option is given.

Running a custom pre-processor at compile-time is in some settings appropriate and useful. The `-F` option lets you run a pre-processor as part of the overall GHC compilation pipeline, which has the advantage over running a Haskell pre-processor separately in that it works in interpreted mode and you can continue to take reap the benefits of GHC's recompilation checker.

The pre-processor is run just before the Haskell compiler proper processes the Haskell input, but after the literate markup has been stripped away and (possibly) the C pre-processor has washed the Haskell input.

Use `-pgmF cmd` 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. 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 l1 -t utf-8 $2 ) > $3
```

and pass `-F -pgmF convert.sh` to GHC. The `-f l1` option tells iconv to convert your Latin-1 file, supplied in argument `$2`, while the "-t utf-8" options tell iconv to return a UTF-8 encoded file. The result is redirected into argument `$3`. The `echo "{-#LINE 1 \"$2\" #-}"` just makes sure that your error positions are reported as in the original source file.

4.12.5 Options affecting code generation

-fasm Use GHC's native code generatorrather than compiling via LLVM. `-fasm` is the default.

-fllvm Compile via LLVMinstead 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 to be on the path.

-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`, 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`.

-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.

-dynamic When generating code, assume that entities imported from a different package will reside in a different shared library or binary.

Note that using this option when linking causes GHC to link against shared libraries.

4.12.6 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).

`-llib` Link in the *`lib`* library. On Unix systems, this will be in a file called `lib`*`lib`*`.a` or `lib`*`lib`*`.so` which resides somewhere on the library directories path.

Because of the sad state of most UNIX linkers, the order of such options does matter. If library *`foo`* requires library *`bar`*, then in general `-l`*`foo`* should come *before* `-l`*`bar`* on the command line.

There's one other gotcha to bear in mind when using external libraries: if the library contains a `main()` function, then this will be linked in preference to GHC's own `main()` function (eg. `libf2c` and `libl` have their own `main()`s). This is because GHC's `main()` comes from the `HSrts` library, which is normally included *after* all the other libraries on the linker's command line. To force GHC's `main()` to be used in preference to any other `main()`s from external libraries, just add the option `-lHSrts` before any other libraries on the command line.

`-c` Omits the link step. This option can be used with `--make` 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 Section 4.9), 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` On Darwin/OS X/iOS only, link all passed files into a static library suitable for linking into an iOS (when using a cross-compiler) or Mac Xcode project. To control the name, use the `-o` *`name`* option as usual. The default name is `liba.a`. This should nearly always be passed when compiling for iOS with a cross-compiler.

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

`-framework-pathdir` 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. We use this feature for building GHC's libraries (warning: don't use it unless you know what you're doing!).

`-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 Section 4.13 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`/`-static` control whether the resulting shared object links statically or dynamically to Haskell package libraries given as `-package` 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`, see Section 4.12.5

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 Section 4.13.4 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` 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 Section 8.2.1.1.

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` 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` mode when there is no Haskell `Main` module present (normally the compiler will not attempt linking when there is no `Main`).

The flags `-rtsopts` and `-with-rtsopts` have no effect when used with `-no-hs-main`, because they are implemented by changing the definition of `main` that GHC generates. See Section 8.2.1.1 for how to get the effect of `-rtsopts` and `-with-rtsopts` 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` RTS option to be used, which allows threads to run in parallel on a multiprocessor or multicore machine. See Section 4.15.
- 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 export`ed Haskell functions may be called from multiple OS threads simultaneously. See Section 8.2.4.

-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 Section 4.17.6 for more information.

`-eventlog` can be used with `-threaded`. It is implied by `-debug`.

-rtsopts This option affects the processing of RTS control options given either on the command line or via the `GHCRTS` 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=some [this is the default setting] Enable only the "safe" RTS options: (Currently only `-?` and `--info`.) Any other RTS options on the command line or in the `GHCRTS` environment variable causes the program with to abort with an error message.

-rtsopts=all, or just -rtsopts Enable *all* RTS option processing, both on the command line and through the `GHCRTS` environment variable.

In GHC 6.12.3 and earlier, the default was to process all RTS options. However, since RTS options can be used to write logging data to arbitrary files under the security context of the running program, there is a potential security problem. For this reason, GHC 7.0.1 and later default to `-rtsops=some`.

Note that `-rtsopts` has no effect when used with `-no-hs-main`; see Section 8.2.1.1 for details.

-with-rtsopts 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 Section 8.2.1.1 for details.

-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` also implies `-fno-embed-manifest`, 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` flag. A GHC installation comes with its own copy of `windres` for this reason.

See also `-pgmwindres` (Section 4.12.1) and `-optwindres` (Section 4.12.2).

-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` 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` 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.

4.13 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 Section 12.6), 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.

4.13.1 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` 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).

4.13.2 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`, 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 Section 4.9.8. 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`, `-fPIC` and `-shared`.

4.13.3 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 Section 8.2.1.2. 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`, `-fPIC` and `-shared` flags:

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

As before, the `-dynamic` flag specifies that this library links against the shared library versions of the rts and base package. The `-fPIC` flag is required for all code that will end up in a shared library. The `-shared` 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` without `-dynamic` in the link step. That means to statically link the rts all 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` 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.

4.13.4 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.

4.13.4.1 Unix

On Unix there are two mechanisms. Shared libraries can be installed into standard locations that the dynamic linker knows about. For example `/usr/lib` or `/usr/local/lib` on most systems. The other mechanism is to use a "runtime path" or "rpath" embedded into programs and libraries themselves. These paths can either be absolute paths or on at least Linux and Solaris they can be paths relative to the program or library itself. In principle this makes it possible to construct fully relocatable sets of programs and libraries.

GHC has a `-dynload` linking flag to select the method that is used to find shared libraries at runtime. There are currently two modes:

sysdep A system-dependent mode. This is also the default mode. On Unix ELF systems this embeds `RPATH`/`RUNPATH` entries into the shared library or executable. In particular it uses absolute paths to where the shared libraries for the rts and each package can be found. This means the program can immediately be run and it will be able to find the libraries it needs. However it may not be suitable for deployment if the libraries are installed in a different location on another machine.

deploy This does not embed any runtime paths. It relies on the shared libraries being available in a standard location or in a directory given by the `LD_LIBRARY_PATH` environment variable.

To use relative paths for dependent libraries on Linux and Solaris you can pass a suitable `-rpath` flag to the linker:

```
ghc -dynamic Main.hs -o main -lfoo -L. -optl-Wl,-rpath,'$ORIGIN'
```

This assumes that the library `libfoo.so` is in the current directory and will be able to be found in the same directory as the executable `main` once the program is deployed. Similarly it would be possible to use a subdirectory relative to the executable e.g. `-optl-Wl,-rpath,'$ORIGIN/lib'`.

This relative path technique can be used with either of the two `-dynload` modes, though it makes most sense with the `deploy` mode. The difference is that with the `deploy` mode, the above example will end up with an ELF `RUNPATH` of just `$ORIGIN` while with the `sysdep` mode the `RUNPATH` will be `$ORIGIN` followed by all the library directories of all the packages that the program depends on (e.g. `base` and `rts` packages etc.) which are typically absolute paths. The unix tool `readelf --dynamic` is handy for inspecting the `RPATH`/`RUNPATH` entries in ELF shared libraries and executables.

4.13.4.2 Mac OS X

The standard assumption on Darwin/Mac OS X is that dynamic libraries will be stamped at build time with an "install name", which is the full ultimate install path of the library file. Any libraries or executables that subsequently link against it (even if it hasn't been installed yet) will pick up that path as their runtime search location for it. When compiling with ghc directly, the install name is set by default to the location where it is built. You can override this with the `-dylib-install-name` 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.

4.14 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` option (see Section 4.12.6. This provides two benefits:

- It enables the `-N` RTS option to be used, which allows threads to run in parallel on a multiprocessor or multicore machine. See Section 4.15.
- 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 export`ed Haskell functions may be called from multiple OS threads simultaneously. See Section 8.2.4.

The following RTS option(s) affect the behaviour of Concurrent Haskell programs:

`-C`*`s`* 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). By default, context switches occur every 20ms.

4.15 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 Section 7.28 we describe the language features that affect parallelism.

4.15.1 Compile-time options for SMP parallelism

In order to make use of multiple CPUs, your program must be linked with the `-threaded` option (see Section 4.12.6). Additionally, the following compiler options affect parallelism:

`-feager-blackholing` Blackholing is the act of marking a thunk (lazy computuation) 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` 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` flag.

4.15.2 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` option.

`-N[x]` Use *x* simultaneous threads when running the program. Normally *x* should be chosen to match the number of CPU cores on the machine[3]. 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.

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.

Setting `-N` also has the effect of enabling the parallel garbage collector (see Section 4.17.3).

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. This is an experimental feature, and may or may not be useful. Please let us know whether it helps for you!

`-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`.

4.15.3 Hints for using SMP parallelism

Add the `-s` 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` for comparison.

The output of `+RTS -s` tells you how many "sparks" were created and executed during the run of the program (see Section 4.17.3), 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.

[3] 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.

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

4.17 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 (Section 4.17.1) describes the various methods, and the following sections describe the RTS options themselves.

4.17.1 Setting RTS options

There are four ways to set RTS options:

- on the command line between `+RTS ... -RTS`, when running the program (Section 4.17.1.1)
- at compile-time, using `--with-rtsopts` (Section 4.17.1.2)
- with the environment variable `GHCRTS` (Section 4.17.1.3)
- by overriding "hooks" in the runtime system (Section 4.17.1.4)

4.17.1.1 Setting RTS options on the command line

If you set the `-rtsopts` flag appropriately when linking (see Section 4.12.6), 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 -?` 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. eg. to set the maximum heap size for a compilation to 128M, you would add `+RTS -M128m -RTS` to the command line.

4.17.1.2 Setting RTS options at compile time

GHC lets you change the default RTS options for a program at compile time, using the `-with-rtsopts` flag (Section 4.12.6). 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"`.

4.17.1.3 Setting RTS options with the GHCRTS environment variable

If the `-rtsopts` flag is set to something other than `none` when linking, RTS options are also taken from the environment variable `GHCRTS`. 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` 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).

4.17.1.4 "Hooks" to change RTS behaviour

GHC lets you exercise rudimentary control over certain RTS settings for any given program, by compiling in a "hook" that is called by the run-time system. The RTS contains stub definitions for these hooks, but by writing your own version and linking it on the GHC command line, you can override the defaults.

Owing to the vagaries of DLL linking, these hooks don't work under Windows when the program is built dynamically.

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.

4.17.2 Miscellaneous RTS options

`-V`*`secs`* Sets the interval that the RTS clock ticks at. The runtime uses a single timer signal to count ticks; this timer signal is used to control the context switch timer (Section 4.14) and the heap profiling timer Section 5.4.1. Also, the time profiler uses the RTS timer signal directly to record time profiling samples.

Normally, setting the `-V` option directly is not necessary: the resolution of the RTS timer is adjusted automatically if a short interval is requested with the `-C` or `-i` options. However, setting `-V` 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.

`--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 above).

`-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 -xm20000000 -RTS` would hint that the RTS should allocate starting at the 0.5Gb mark. The default is to use the OS's built-in support for allocating memory in the low 2Gb if available (e.g. `mmap` with `MAP_32BIT` on Linux), or otherwise `-xm40000000`.

`-xq`*`size`* [Default: 100k] This option relates to allocation limits; for more about this see `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.

4.17.3 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: 512k] 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`, 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 (`-G1`) 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`, below).

`-n`*`size`* [Default: 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` 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 >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` 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*.

-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* than to increase `-F`*factor*.

The `-F` setting will be automatically reduced by the garbage collector when the maximum heap size (the `-M`*size* 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` option (see above) 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` option, see below).

-qg[*gen*] [New in GHC 6.12.1] [Default: 0] 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 `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` option, and additionally it might be beneficial to restrict parallel GC to the old generation with `-qg1`.

-qb[*gen*] [New in GHC 6.12.1] [Default: 1] Use load-balancing in the parallel GC in generation *gen* and higher. Omitting *gen* disables load-balancing entirely.

Load-balancing shares out the work of GC between the available cores. This is a good idea when the heap is large and we need to parallelise the GC work, however it is also pessimal for the short young-generation collections in a parallel program, because it can harm locality by moving data from the cache of the CPU where is it being used to the cache of another CPU. Hence the default is to do load-balancing only in the old-generation. In fact, for a parallel program it is sometimes beneficial to disable load-balancing entirely with `-qb`.

-H[*size*] [Default: 0] This option provides a "suggested heap size" for the garbage collector. Think of `-H`*size* as a variable `-A` 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) In the threaded and SMP versions of the RTS (see `-threaded`, Section 4.12.6), 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. (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).

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.

-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` 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`), because in a chain of stack chunks, each chunk will have a gap of unused space of this size.

-K*size* [Default: 80% physical memory size] 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* Minimum % *`n`* of heap which must be available for allocation. The default is 3%.

-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.

-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, 0.00 INIT (0.00 elapsed), 0.02 MUT (0.02 elapsed), 0.07 GC (0.07 elapsed) : ↩
    ghc>>
```

This tells you:

- The total number of bytes allocated by the program over the whole run.
- The total number of garbage collections performed.

- The average and maximum "residency", which is the amount of live data in bytes. The runtime can only determine the amount of live data during a major GC, which is why the number of samples corresponds to the number of major GCs (and is usually relatively small). To get a better picture of the heap profile of your program, use the `-hT` RTS option (Section 4.17.5).
- 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    Copied     Live    GC    GC     TOT     TOT  Page Flts
    bytes     bytes     bytes  user  elap    user    elap
   528496     47728    141512  0.01  0.02    0.02    0.02    0    0  (Gen:  1)
[...]
   524944    175944   1726384  0.00  0.00    0.08    0.11    0    0  (Gen:  0)
```

For each garbage collection, we print:

- How many bytes we allocated this garbage collection.
- How many bytes we copied this garbage collection.
- How many bytes are currently live.
- How long this garbage collection took (CPU time and elapsed wall clock time).
- How long the program has been running (CPU time and elapsed wall clock time).
- How many page faults occurred this garbage collection.
- How many page faults occurred since the end of the last garbage collection.
- Which generation is being garbage collected.

4.17.4 RTS options for concurrency and parallelism

The RTS options related to concurrency are described in Section 4.14, and those for parallelism in Section 4.15.2.

4.17.5 RTS options for profiling

Most profiling runtime options are only available when you compile your program for profiling (see Section 5.2, and Section 5.4.1 for the runtime options). However, there is one profiling option that is available for ordinary non-profiled executables:

`-hT` (can be shortened to `-h`.) Generates a basic heap profile, in the file *`prog`*`.hp`. To produce the heap profile graph, use **hp2ps** (see Section 5.5). 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 (Chapter 5).

4.17.6 Tracing

When the program is linked with the `-eventlog` option (Section 4.12.6), runtime events can be logged in two 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.
- As text to standard output, for debugging purposes.

`-l[flags]` Log events in binary format to the file *`program`*`.eventlog`. Without any *`flags`* specified, this logs a default set of events, suitable for use with tools like ThreadScope.

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`. Enable

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 or `-a` for 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.

4.17.7 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..."

`-Dx` An RTS debugging flag; only available if the program was linked with the `-debug` 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` 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`). The *`file`* business works just like on the `-S` RTS option, above.

For more information on ticky-ticky profiling, see Section 5.8.

-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` 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.)

4.17.8 Getting information about the RTS

It is possible to ask the RTS to give some information about itself. To do this, use the `--info` flag, e.g.

```
$ ./a.out +RTS --info
 [("GHC RTS", "YES")
 ,("GHC version", "6.7")
 ,("RTS way", "rts_p")
 ,("Host platform", "x86_64-unknown-linux")
 ,("Host architecture", "x86_64")
 ,("Host OS", "linux")
 ,("Host vendor", "unknown")
 ,("Build platform", "x86_64-unknown-linux")
 ,("Build architecture", "x86_64")
 ,("Build OS", "linux")
 ,("Build vendor", "unknown")
 ,("Target platform", "x86_64-unknown-linux")
 ,("Target architecture", "x86_64")
 ,("Target OS", "linux")
 ,("Target vendor", "unknown")
 ,("Word size", "64")
```

```
,("Compiler unregisterised", "NO")
,("Tables next to code", "YES")
]
```

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` option) and `rts_p` (profiling runtime, i.e. linked using the `-prof` option). Other variants include `debug` (linked using `-debug`), 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" 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.)

4.18 Debugging the compiler

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

4.18.1 Dumping out compiler intermediate structures

`-ddump-`*`pass`* Make a debugging dump after pass `<pass>` (may be common enough to need a short form. . .). You can get all of these at once (*lots* of output) by using `-v5`, or most of them with `-v4`. You can prevent them from clogging up your standard output by passing `-ddump-to-file`. Some of the most useful ones are:

- **`-ddump-parsed`:** parser output
- **`-ddump-rn`:** renamer output
- **`-ddump-tc`:** typechecker output
- **`-ddump-splices`:** Dump Template Haskell expressions that we splice in, and what Haskell code the expression evaluates to.
- **`-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` dumps a type signature for all the imported and system-defined things as well; useful for debugging the compiler.
- **`-ddump-deriv`:** derived instances

-ddump-ds: desugarer output

-ddump-spec: output of specialisation pass

-ddump-rules: dumps all rewrite rules specified in this module; see Section 7.23.7.

-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: simplifier output (Core-to-Core passes)

-ddump-inlinings: inlining info from the simplifier

-ddump-cpranal: CPR analyser output

-ddump-stranal: strictness analyser output

-ddump-strsigs: strictness signatures

-ddump-cse: CSE pass output

-ddump-worker-wrapper: worker/wrapper split output

-ddump-occur-anal: `occurrence analysis' output

-ddump-prep: output of core preparation pass

-ddump-stg: output of STG-to-STG passes

-ddump-flatC: *flattened* Abstract C

-ddump-cmm: Print the C-- code out.

-ddump-opt-cmm: Dump the results of C-- to C-- optimising passes.

-ddump-asm: assembly language from the native code generator

-ddump-llvm: LLVM code from the LLVM code generator

-ddump-bcos: byte code compiler output

-ddump-foreign: dump foreign export stubs

-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 too place. If you add `-dppr-debug` you get more detailed information.

-ddump-if-trace Make the interface loader be *real* chatty about what it is up to.

-ddump-tc-trace Make the type checker be *real* chatty about what it is up to.

-ddump-vt-trace Make the vectoriser 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-rn-stats Print out summary of what kind of information the renamer had to bring in.

-dverbose-core2core, -dverbose-stg2stg Show the output of the intermediate Core-to-Core and STG-to-STG passes, respectively. (*Lots* of output!) So: when we're really desperate:

```
% ghc -noC -O -ddump-simpl -dverbose-core2core -dcore-lint Foo.hs
```

-dshow-passes Print out each pass name as it happens.

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

-dfaststring-stats Show statistics for 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 foralls, and variables have their unique-id attached (so you can check for things that look the same but aren't). This flag makes debugging output appear in the more verbose debug style.

4.18.2 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-colsNNN Set the width of debugging output. Use this if your code is wrapping too much. For example: `-dppr-cols200`.

-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.

4.18.3 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-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`

-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-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.

4.18.4 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.

4.18.5 How to read Core syntax (from some -ddump flags)

Let's do this by commenting an example. It's from doing `-ddump-ds` on this code:

```
skip2 m = m : skip2 (m+2)
```

Before we jump in, a word about names of things. Within GHC, variables, type constructors, etc., are identified by their "Uniques." These are of the form `letter' plus `number' (both loosely interpreted). The `letter' gives some idea of where the Unique came from; e.g., _ means "built-in type variable"; `t` means "from the typechecker"; `s` means "from the simplifier"; and so on. The `number' is printed fairly compactly in a `base-62' format, which everyone hates except me (WDP).

Remember, everything has a "Unique" and it is usually printed out when debugging, in some form or another. So here we go...

```
Desugared:
Main.skip2{-r1L6-} :: _forall_ a$_4 =>{{Num a$_4}} -> a$_4 -> [a$_4]

--# `r1L6' is the Unique for Main.skip2;
--# `_4' is the Unique for the type-variable (template) `a'
--# `{{Num a$_4}}' is a dictionary argument

_NI_

--# `_NI_' means "no (pragmatic) information" yet; it will later
--# evolve into the GHC_PRAGMA info that goes into interface files.

Main.skip2{-r1L6-} =
    /\ _4 -> \ d.Num.t4Gt ->
        let {
          {- CoRec -}
          +.t4Hg :: _4 -> _4 -> _4
          _NI_
          +.t4Hg = (+{-r3JH-} _4) d.Num.t4Gt

          fromInt.t4GS :: Int{-2i-} -> _4
          _NI_
          fromInt.t4GS = (fromInt{-r3JX-} _4) d.Num.t4Gt

--# The `+' class method (Unique: r3JH) selects the addition code
--# from a `Num' dictionary (now an explicit lambda'd argument).
--# Because Core is 2nd-order lambda-calculus, type applications
--# and lambdas (/\) are explicit.  So `+' is first applied to a
--# type (`_4'), then to a dictionary, yielding the actual addition
--# function that we will use subsequently...

--# We play the exact same game with the (non-standard) class method
--# `fromInt'.  Unsurprisingly, the type `Int' is wired into the
--# compiler.

          lit.t4Hb :: _4
          _NI_
          lit.t4Hb =
              let {
                ds.d4Qz :: Int{-2i-}
                _NI_
                ds.d4Qz = I#! 2#
              } in  fromInt.t4GS ds.d4Qz

--# `I# 2#' is just the literal Int `2'; it reflects the fact that
--# GHC defines `data Int = I# Int#', where Int# is the primitive
--# unboxed type.  (see relevant info about unboxed types elsewhere...)

--# The `!' after `I#' indicates that this is a *saturated*
--# application of the `I#' data constructor (i.e., not partially
--# applied).

          skip2.t3Ja :: _4 -> [_4]
          _NI_
          skip2.t3Ja =
              \ m.r1H4 ->
                  let { ds.d4QQ :: [_4]
                        _NI_
                        ds.d4QQ =
                    let {
                      ds.d4QY :: _4
```

```
                      _NI_
                      ds.d4QY = +.t4Hg m.r1H4 lit.t4Hb
                    } in  skip2.t3Ja ds.d4QY
                  } in
                  :! _4 m.r1H4 ds.d4QQ

          {- end CoRec -}
        } in  skip2.t3Ja
```

("It's just a simple functional language" is an unregisterised trademark of Peyton Jones Enterprises, plc.)

4.19 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 Section 4.3), and the flag's opposite (if available).

4.19.1 Verbosity options

More details in Section 4.6

Flag	Description	Static/Dynamic	Reverse
`-v`	verbose mode (equivalent to `-v3`)	dynamic	-
`-v`*n*	set verbosity level	dynamic	-
`-fprint-explicit-foralls`	print explicit `forall` quantification in types	dynamic	-fno-print-explicit-foralls
`-fprint-explicit-kinds`	print explicit kind foralls and kind arguments in types	dynamic	-fno-print-explicit-kinds
`-ferror-spans`	output full span in error messages	dynamic	-
`-H`*size*	Set the minimum heap size to *size*	dynamic	-
`-Rghc-timing`	Summarise timing stats for GHC (same as `+RTS -tstderr`)	dynamic	-

4.19.2 Alternative modes of operation

More details in Section 4.5

Flag	Description	Static/Dynamic	Reverse
`--help,-?`	Disply help	mode	-
`--interactive`	Interactive mode - normally used by just running **ghci**; see Chapter 2 for details.	mode	-
`--make`	Build a multi-module Haskell program, automatically figuring out dependencies. Likely to be much easier, and faster, than using **make**; see Section 4.5.1 for details..	mode	-
`-e` *expr*	Evaluate *expr*; see Section 4.5.2 for details.	mode	-

Flag	Description	Static/Dynamic	Reverse
`--show-iface`	Display the contents of an interface file.	mode	-
`-M`	Generate dependency information suitable for use in a `Makefile`; see Section 4.7.12 for details.	mode	-
`--supported-extensions`, `--supported-languages`	display the supported language extensions	mode	-
`--show-options`	display the supported command line options	mode	-
`--info`	display information about the compiler	mode	-
`--version, -V`	display GHC version	mode	-
`--numeric-version`	display GHC version (numeric only)	mode	-
`--print-libdir`	display GHC library directory	mode	-

4.19.3 Which phases to run

Section 4.5.3

Flag	Description	Static/Dynamic	Reverse
`-E`	Stop after preprocessing (`.hspp` file)	mode	-
`-C`	Stop after generating C (`.hc` file)	mode	-
`-S`	Stop after generating assembly (`.s` file)	mode	-
`-c`	Stop after generating object (`.o`) file	mode	-
`-x` *suffix*	Override default behaviour for source files	dynamic	-

4.19.4 Redirecting output

Section 4.7.4

Flag	Description	Static/Dynamic	Reverse
`-hcsuf` *suffix*	set the suffix to use for intermediate C files	dynamic	-
`-hidir` *dir*	set directory for interface files	dynamic	-
`-hisuf` *suffix*	set the suffix to use for interface files	dynamic	-
`-o` *filename*	set output filename	dynamic	-
`-odir` *dir*	set directory for object files	dynamic	-
`-ohi` *filename*	set the filename in which to put the interface	dynamic	
`-osuf` *suffix*	set the output file suffix	dynamic	-
`-stubdir` *dir*	redirect FFI stub files	dynamic	-
`-dumpdir` *dir*	redirect dump files	dynamic	-

Flag	Description	Static/Dynamic	Reverse
`-outputdir` *dir*	set output directory	dynamic	-

4.19.5 Keeping intermediate files

Section 4.7.5

Flag	Description	Static/Dynamic	Reverse
`-keep-hc-file` or `-keep-hc-files`	retain intermediate `.hc` files	dynamic	-
`-keep-llvm-file` or `-keep-llvm-files`	retain intermediate LLVM `.ll` files	dynamic	-
`-keep-s-file` or `-keep-s-files`	retain intermediate `.s` files	dynamic	-
`-keep-tmp-files`	retain all intermediate temporary files	dynamic	-

4.19.6 Temporary files

Section 4.7.6

Flag	Description	Static/Dynamic	Reverse
`-tmpdir`	set the directory for temporary files	dynamic	-

4.19.7 Finding imports

Section 4.7.3

Flag	Description	Static/Dynamic	Reverse
`-i`*dir1*:*dir2*:...	add *dir*, *dir2*, etc. to import path	dynamic/`:set`	-
`-i`	Empty the import directory list	dynamic/`:set`	-

4.19.8 Interface file options

Section 4.7.7

Flag	Description	Static/Dynamic	Reverse
`-ddump-hi`	Dump the new interface to stdout	dynamic	-
`-ddump-hi-diffs`	Show the differences vs. the old interface	dynamic	-
`-ddump-minimal-imports`	Dump a minimal set of imports	dynamic	-
`--show-iface` *file*	See Section 4.5.		

4.19.9 Recompilation checking

Section 4.7.8

Flag	Description	Static/Dynamic	Reverse
-fforce-recomp	Turn off recompilation checking. This is implied by any -ddump-X option when compiling a single file (i.e. when using -c).	dynamic	-fno-force-recomp

4.19.10 Interactive-mode options

Section 2.9

Flag	Description	Static/Dynamic	Reverse
-ignore-dot-ghci	Disable reading of .ghci files	dynamic	-
-ghci-script	Read additional .ghci files	dynamic	-
-fbreak-on-error	Break on uncaught exceptions and errors	dynamic	-fno-break-on-error
-fbreak-on-exception	Break on any exception thrown	dynamic	-fno-break-on-exception
-fghci-hist-size=*n*	Set the number of entries GHCi keeps for :history	dynamic	(default is 50)
-fprint-evld-with-show	Enable usage of Show instances in :print	dynamic	-fno-print-evld-with-show
-fprint-bind-result	Turn on printing of binding results in GHCi	dynamic	-fno-print-bind-result
-fno-print-bind-contents	Turn off printing of binding contents in GHCi	dynamic	-
-fno-implicit-import-qualified	Turn off implicit qualified import of everything in GHCi	dynamic	-
-interactive-print	Select the function to use for printing evaluated expressions in GHCi	dynamic	-

4.19.11 Packages

Section 4.9

Flag	Description	Static/Dynamic	Reverse
-this-package-key *P*	Compile to be part of package *P*	dynamic	-
-package *P*	Expose package *P*	dynamic/:set	-
-hide-all-packages	Hide all packages by default	dynamic	-
-hide-package *name*	Hide package *P*	dynamic/:set	-
-ignore-package *name*	Ignore package *P*	dynamic/:set	-
-package-db *file*	Add *file* to the package db stack.	dynamic	-
-clear-package-db	Clear the package db stack.	dynamic	-
-no-global-package-db	Remove the global package db from the stack.	dynamic	-
-global-package-db	Add the global package db to the stack.	dynamic	-
-no-user-package-db	Remove the user's package db from the stack.	dynamic	-
-user-package-db	Add the user's package db to the stack.	dynamic	-

Flag	Description	Static/Dynamic	Reverse
-no-auto-link-packages	Don't automatically link in the base and rts packages.	dynamic	-
-trust *P*	Expose package *P* and set it to be trusted	dynamic/:set	-
-distrust *P*	Expose package *P* and set it to be distrusted	dynamic/:set	-
-distrust-all	Distrust all packages by default	dynamic/:set	-

4.19.12 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 Section 7.1. Some options are enabled using `-f*` flags.

Flag	Description	Static/Dynamic	Reverse
-fcontext-stack=*n*	set the limit for type-class context reduction. Default is 100.	dynamic	
-fglasgow-exts	Deprecated. Enable most language extensions; see Section 7.1 for exactly which ones.	dynamic	-fno-glasgow-exts
-firrefutable-tuples	Make tuple pattern matching irrefutable	dynamic	-fno-irrefutable-tuples
-fpackage-trust	Enable Safe Haskell trusted package requirement for trustworthy modules.	dynamic	-
-ftype-function-depth=*n*	set the limit for type function reductions. Default is 200.	dynamic	
-XAllowAmbiguousTypes	Allow the user to write ambiguous types, and the type inference engine to infer them.	dynamic	-XNoAllowAmbiguousTypes
-XArrows	Enable arrow notation extension	dynamic	-XNoArrows
-XAutoDeriveTypeable	Automatically derive Typeable instances for every datatype and type class declaration. Implies -XDeriveDataTypeable.	dynamic	-XNoAutoDeriveTypeable
-XBangPatterns	Enable bang patterns.	dynamic	-XNoBangPatterns
-XBinaryLiterals	Enable support for binary literals.	dynamic	-XNoBinaryLiterals
-XCApiFFI	Enable the CAPI calling convention.	dynamic	-XNoCAPIFFI
-XConstrainedClassMethods	Enable constrained class methods.	dynamic	-XNoConstrainedClassMethods
-XConstraintKinds	Enable a kind of constraints.	dynamic	-XNoConstraintKinds
-XCPP	Enable the C preprocessor.	dynamic	-XNoCPP
-XDataKinds	Enable datatype promotion.	dynamic	-XNoDataKinds
-XDefaultSignatures	Enable default signatures.	dynamic	-XNoDefaultSignatures
-XDeriveAnyClass	Enable deriving for any class.	dynamic	-XNoDeriveAnyClass

Flag	Description	Static/Dynamic	Reverse
-XDeriveDataTypeable	Enable deriving for the Data and Typeable classes. Implied by -XAutoDeriveTypeable.	dynamic	-XNoDeriveDataTypeable
-XDeriveFunctor	Enable deriving for the Functor class. Implied by -XDeriveTraversable.	dynamic	-XNoDeriveFunctor
-XDeriveFoldable	Enable deriving for the Foldable class. Implied by -XDeriveTraversable.	dynamic	-XNoDeriveFoldable
-XDeriveGeneric	Enable deriving for the Generic class.	dynamic	-XNoDeriveGeneric
-XDeriveTraversable	Enable deriving for the Traversable class. Implies -XDeriveFunctor and -XDeriveFoldable.	dynamic	-XNoDeriveTraversable
-XDisambiguateRecordFields	Enable record field disambiguation. Implied by -XRecordWildCards.	dynamic	-XNoDisambiguateRecordFields
-XEmptyCase	Allow empty case alternatives.	dynamic	-XNoEmptyCase
-XEmptyDataDecls	Enable empty data declarations.	dynamic	-XNoEmptyDataDecls
-XExistentialQuantification	Enable existential quantification.	dynamic	-XNoExistentialQuantification
-XExplicitForAll	Enable explicit universal quantification. Implied by -XScopedTypeVariables, -XLiberalTypeSynonyms, -XRankNTypes and -XExistentialQuantification.	dynamic	-XNoExplicitForAll
-XExplicitNamespaces	Enable using the keyword type to specify the namespace of entries in imports and exports (Section 7.3.26.4). Implied by -XTypeOperators and -XTypeFamilies.	dynamic	-XNoExplicitNamespaces
-XExtendedDefaultRules	Use GHCi's extended default rules in a normal module.	dynamic	-XNoExtendedDefaultRules
-XFlexibleContexts	Enable flexible contexts. Implied by -XImplicitParams.	dynamic	-XNoFlexibleContexts
-XFlexibleInstances	Enable flexible instances. Implies -XTypeSynonymInstances. Implied by -XImplicitParams.	dynamic	-XNoFlexibleInstances
-XForeignFunctionInterface	Enable foreign function interface.	dynamic	-XNoForeignFunctionInterface

Flag	Description	Static/Dynamic	Reverse
-XFunctionalDependencies	Enable functional dependencies. Implies -XMultiParamTypeClasses.	dynamic	-XNoFunctionalDependencies
-XGADTs	Enable generalised algebraic data types. Implies -XGADTSyntax and -XMonoLocalBinds.	dynamic	-XNoGADTs
-XGADTSyntax	Enable generalised algebraic data type syntax.	dynamic	-XNoGADTSyntax
-XGeneralizedNewtypeDeriving	Enable newtype deriving.	dynamic	-XNoGeneralizedNewtypeDeriving
-XGenerics	Deprecated, does nothing. No longer enables generic classes. See also GHC's support for generic programming.	dynamic	-XNoGenerics
-XImplicitParams	Enable Implicit Parameters. Implies -XFlexibleContexts and -XFlexibleInstances.	dynamic	-XNoImplicitParams
-XNoImplicitPrelude	Don't implicitly import Prelude. Implied by -XRebindableSyntax.	dynamic	-XImplicitPrelude
-XImpredicativeTypes	Enable impredicative types. Implies -XRankNTypes.	dynamic	-XNoImpredicativeTypes
-XIncoherentInstances	Enable incoherent instances. Implies -XOverlappingInstances.	dynamic	-XNoIncoherentInstances
-XInstanceSigs	Enable instance signatures.	dynamic	-XNoInstanceSigs
-XInterruptibleFFI	Enable interruptible FFI.	dynamic	-XNoInterruptibleFFI
-XKindSignatures	Enable kind signatures. Implied by -XTypeFamilies and -XPolyKinds.	dynamic	-XNoKindSignatures
-XLambdaCase	Enable lambda-case expressions.	dynamic	-XNoLambdaCase
-XLiberalTypeSynonyms	Enable liberalised type synonyms.	dynamic	-XNoLiberalTypeSynonyms
-XMagicHash	Allow "#" as a postfix modifier on identifiers.	dynamic	-XNoMagicHash
-XMonadComprehensions	Enable monad comprehensions.	dynamic	-XNoMonadComprehensions
-XMonoLocalBinds	Enable do not generalise local bindings. Implied by -XTypeFamilies and -XGADTs.	dynamic	-XNoMonoLocalBinds
-XNoMonomorphismRestriction	Disable the monomorphism restriction.	dynamic	-XMonomorphismRestriction

Flag	Description	Static/Dynamic	Reverse
`-XMultiParamTypeClasses`	Enable multi parameter type classes. Implied by `-XFunctionalDependencies`.	dynamic	`-XNoMultiParamTypeClasses`
`-XMultiWayIf`	Enable multi-way if-expressions.	dynamic	`-XNoMultiWayIf`
`-XNamedFieldPuns`	Enable record puns.	dynamic	`-XNoNamedFieldPuns`
`-XNamedWildcards`	Enable named wildcards.	dynamic	`-XNoNamedWildcards`
`-XNegativeLiterals`	Enable support for negative literals.	dynamic	`-XNoNegativeLiterals`
`-XNoNPlusKPatterns`	Disable support for `n+k` patterns.	dynamic	`-XNPlusKPatterns`
`-XNullaryTypeClasses`	Deprecated, does nothing. nullary (no parameter) type classes are now enabled using `-XMultiParamTypeClasses`.	dynamic	`-XNoNullaryTypeClasses`
`-XNumDecimals`	Enable support for 'fractional' integer literals.	dynamic	`-XNoNumDecimals`
`-XOverlappingInstances`	Enable overlapping instances.	dynamic	`-XNoOverlappingInstances`
`-XOverloadedLists`	Enable overloaded lists.	dynamic	`-XNoOverloadedLists`
`-XOverloadedStrings`	Enable overloaded string literals.	dynamic	`-XNoOverloadedStrings`
`-XPackageImports`	Enable package-qualified imports.	dynamic	`-XNoPackageImports`
`-XParallelArrays`	Enable parallel arrays. Implies `-XParallelListComp`.	dynamic	`-XNoParallelArrays`
`-XParallelListComp`	Enable parallel list comprehensions. Implied by `-XParallelArrays`.	dynamic	`-XNoParallelListComp`
`-XPartialTypeSignatures`	Enable partial type signatures.	dynamic	`-XNoPartialTypeSignatures`
`-XPatternGuards`	Enable pattern guards.	dynamic	`-XNoPatternGuards`
`-XPatternSynonyms`	Enable pattern synonyms.	dynamic	`-XNoPatternSynonyms`
`-XPolyKinds`	Enable kind polymorphism. Implies `-XKindSignatures`.	dynamic	`-XNoPolyKinds`
`-XPolymorphicComponents`	Enable polymorphic components for data constructors.	dynamic, synonym for `-XRankNTypes`	`-XNoPolymorphicComponents`
`-XPostfixOperators`	Enable postfix operators.	dynamic	`-XNoPostfixOperators`
`-XQuasiQuotes`	Enable quasiquotation.	dynamic	`-XNoQuasiQuotes`
`-XRank2Types`	Enable rank-2 types.	dynamic, synonym for `-XRankNTypes`	`-XNoRank2Types`
`-XRankNTypes`	Enable rank-N types. Implied by `-XImpredicativeTypes`.	dynamic	`-XNoRankNTypes`
`-XRebindableSyntax`	Employ rebindable syntax. Implies `-XNoImplicitPrelude`.	dynamic	`-XNoRebindableSyntax`

Flag	Description	Static/Dynamic	Reverse
-XRecordWildCards	Enable record wildcards. Implies -XDisambiguateRecordFields.	dynamic	-XNoRecordWildCards
-XRecursiveDo	Enable recursive do (mdo) notation.	dynamic	-XNoRecursiveDo
-XRelaxedPolyRec	*(deprecated)* Relaxed checking for mutually-recursive polymorphic functions.	dynamic	-XNoRelaxedPolyRec
-XRoleAnnotations	Enable role annotations.	dynamic	-XNoRoleAnnotations
-XSafe	Enable the Safe Haskell Safe mode.	dynamic	-
-XScopedTypeVariables	Enable lexically-scoped type variables.	dynamic	-XNoScopedTypeVariables
-XStandaloneDeriving	Enable standalone deriving.	dynamic	-XNoStandaloneDeriving
-XTemplateHaskell	Enable Template Haskell.	dynamic	-XNoTemplateHaskell
-XNoTraditionalRecordSyntax	Disable support for traditional record syntax (as supported by Haskell 98) C {f =x}	dynamic	-XTraditionalRecordSyntax
-XTransformListComp	Enable generalised list comprehensions.	dynamic	-XNoTransformListComp
-XTrustworthy	Enable the Safe Haskell Trustworthy mode.	dynamic	-
-XTupleSections	Enable tuple sections.	dynamic	-XNoTupleSections
-XTypeFamilies	Enable type families. Implies -XExplicitNamespaces, -XKindSignatures and -XMonoLocalBinds.	dynamic	-XNoTypeFamilies
-XTypeOperators	Enable type operators. Implies -XExplicitNamespaces.	dynamic	-XNoTypeOperators
-XTypeSynonymInstances	Enable type synonyms in instance heads. Implied by -XFlexibleInstances.	dynamic	-XNoTypeSynonymInstances
-XUnboxedTuples	Enable unboxed tuples.	dynamic	-XNoUnboxedTuples
-XUndecidableInstances	Enable undecidable instances.	dynamic	-XNoUndecidableInstances
-XUnicodeSyntax	Enable unicode syntax.	dynamic	-XNoUnicodeSyntax
-XUnliftedFFITypes	Enable unlifted FFI types.	dynamic	-XNoUnliftedFFITypes
-XUnsafe	Enable Safe Haskell Unsafe mode.	dynamic	-
-XViewPatterns	Enable view patterns.	dynamic	-XNoViewPatterns

4.19.13 Warnings

Section 4.8

Flag	Description	Static/Dynamic	Reverse
-W	enable normal warnings	dynamic	-w

Flag	Description	Static/Dynamic	Reverse
`-w`	disable all warnings	dynamic	-
`-Wall`	enable almost all warnings (details in Section 4.8)	dynamic	`-w`
`-Werror`	make warnings fatal	dynamic	-Wwarn
`-Wwarn`	make warnings non-fatal	dynamic	-Werror
`-fdefer-type-errors`	Turn type errors into warnings, deferring the error until runtime. Implies `-fdefer-typed-holes`.	dynamic	`-fno-defer-type-errors`
`-fdefer-typed-holes`	Convert typed hole errors into warnings, deferring the error until runtime. Implied by `-fdefer-type-errors`. See also `-fwarn-typed-holes`.	dynamic	`-fno-defer-typed-holes`
`-fhelpful-errors`	Make suggestions for mis-spelled names.	dynamic	`-fno-helpful-errors`
`-fwarn-deprecated-flags`	warn about uses of commandline flags that are deprecated	dynamic	`-fno-warn-deprecated-flags`
`-fwarn-duplicate-constraints`	warn when a constraint appears duplicated in a type signature	dynamic	`-fno-warn-duplicate-constraints`
`-fwarn-duplicate-exports`	warn when an entity is exported multiple times	dynamic	`-fno-warn-duplicate-exports`
`-fwarn-hi-shadowing`	warn when a `.hi` file in the current directory shadows a library	dynamic	`-fno-warn-hi-shadowing`
`-fwarn-identities`	warn about uses of Prelude numeric conversions that are probably the identity (and hence could be omitted)	dynamic	`-fno-warn-identities`
`-fwarn-implicit-prelude`	warn when the Prelude is implicitly imported	dynamic	`-fno-warn-implicit-prelude`
`-fwarn-incomplete-patterns`	warn when a pattern match could fail	dynamic	`-fno-warn-incomplete-patterns`
`-fwarn-incomplete-uni-patterns`	warn when a pattern match in a lambda expression or pattern binding could fail	dynamic	`-fno-warn-incomplete-uni-patterns`
`-fwarn-incomplete-record-updates`	warn when a record update could fail	dynamic	`-fno-warn-incomplete-record-updates`
`-fwarn-lazy-unlifted-bindings`	*(deprecated)* warn when a pattern binding looks lazy but must be strict	dynamic	`-fno-warn-lazy-unlifted-bindings`
`-fwarn-missing-fields`	warn when fields of a record are uninitialised	dynamic	`-fno-warn-missing-fields`
`-fwarn-missing-import-lists`	warn when an import declaration does not explicitly list all the names brought into scope	dynamic	`-fnowarn-missing-import-lists`
`-fwarn-missing-methods`	warn when class methods are undefined	dynamic	`-fno-warn-missing-methods`

Flag	Description	Static/Dynamic	Reverse
-fwarn-missing-signatures	warn about top-level functions without signatures	dynamic	-fno-warn-missing-signatures
-fwarn-missing-exported-sigs	warn about top-level functions without signatures, only if they are exported. takes precedence over -fwarn-missing-signatures	dynamic	-fno-warn-missing-exported-sigs
-fwarn-missing-local-sigs	warn about polymorphic local bindings without signatures	dynamic	-fno-warn-missing-local-sigs
-fwarn-monomorphism-restriction	warn when the Monomorphism Restriction is applied	dynamic	-fno-warn-monomorphism-restriction
-fwarn-name-shadowing	warn when names are shadowed	dynamic	-fno-warn-name-shadowing
-fwarn-orphans, -fwarn-auto-orphans	warn when the module contains orphan instance declarations or rewrite rules	dynamic	-fno-warn-orphans, -fno-warn-auto-orphans
-fwarn-overlapping-patterns	warn about overlapping patterns	dynamic	-fno-warn-overlapping-patterns
-fwarn-tabs	warn if there are tabs in the source file	dynamic	-fno-warn-tabs
-fwarn-type-defaults	warn when defaulting happens	dynamic	-fno-warn-type-defaults
-fwarn-unrecognised-pragmas	warn about uses of pragmas that GHC doesn't recognise	dynamic	-fno-warn-unrecognised-pragmas
-fwarn-unticked-promoted-constructors	warn if promoted constructors are not ticked	dynamic	-fno-warn-unticked-promoted-constructors
-fwarn-unused-binds	warn about bindings that are unused	dynamic	-fno-warn-unused-binds
-fwarn-unused-imports	warn about unnecessary imports	dynamic	-fno-warn-unused-imports
-fwarn-unused-matches	warn about variables in patterns that aren't used	dynamic	-fno-warn-unused-matches
-fwarn-unused-do-bind	warn about do bindings that appear to throw away values of types other than `()`	dynamic	-fno-warn-unused-do-bind
-fwarn-wrong-do-bind	warn about do bindings that appear to throw away monadic values that you should have bound instead	dynamic	-fno-warn-wrong-do-bind

Flag	Description	Static/Dynamic	Reverse
`-fwarn-unsafe`	warn if the module being compiled is regarded to be unsafe. Should be used to check the safety status of modules when using safe inference. Works on all module types, even those using explicit Safe Haskell modes (such as `-XTrustworthy`) and so can be used to have the compiler check any assumptions made.	dynamic	`-fno-warn-unsafe`
`-fwarn-safe`	warn if the module being compiled is regarded to be safe. Should be used to check the safety status of modules when using safe inference. Works on all module types, even those using explicit Safe Haskell modes (such as `-XTrustworthy`) and so can be used to have the compiler check any assumptions made.	dynamic	`-fno-warn-safe`
`-fwarn-trustworthy-safe`	warn 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.	dynamic	`-fno-warn-safe`
`-fwarn-warnings-deprecations`	warn about uses of functions & types that have warnings or deprecated pragmas	dynamic	`-fno-warn-warnings-deprecations`
`-fwarn-amp`	*(deprecated)* warn on definitions conflicting with the Applicative-Monad Proposal (AMP)	dynamic	`-fno-warn-amp`
`-fwarn-typed-holes`	Report warnings when typed hole errors are deferred until runtime. See `-fdefer-typed-holes`.	dynamic	`-fno-warn-typed-holes`

Flag	Description	Static/Dynamic	Reverse
`-fwarn-partial-type-signatures`	warn about holes in partial type signatures when `-XPartialTypesignatures` is enabled. Not applicable when `-XPartialTypesignatures` is not enabled, in which case errors are generated for such holes. See Section 7.15.	dynamic	`-fno-warn-partial-type-signatures`

4.19.14 Optimisation levels

These options are described in more detail in Section 4.10

Flag	Description	Static/Dynamic	Reverse
`-O0`	Disable optimisations (default)	dynamic	`-O`
`-O` or `-O1`	Enable level 1 optimisations	dynamic	`-O0`
`-O2`	Enable level 2 optimisations	dynamic	`-O0`
`-Odph`	Enable level 2 optimisations, set `-fmax-simplifier-iterations=20` and `-fsimplifier-phases=3`.	dynamic	-

See Section 4.19.15 for a list of optimisations enabled on level 1 and level 2.

4.19.15 Individual optimisations

These options are described in more detail in Section 4.10.2. 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	Static/Dynamic	Reverse
`-fcall-arity`	Enable call-arity optimisation. Implied by `-O`.	dynamic	`-fno-call-arity`
`-fcase-merge`	Enable case-merging. Implied by `-O`.	dynamic	`-fno-case-merge`
`-fcmm-elim-common-blocks`	Enable Cmm common block elimination. Implied by `-O`.	dynamic	`-fno-cmm-elim-common-blocks`
`-fcmm-sink`	Enable Cmm sinking. Implied by `-O`.	dynamic	`-fno-cmm-sink`
`-fcpr-off`	Switch off CPR analysis in the demand analyser.	static	-
`-fcse`	Enable common sub-expression elimination. Implied by `-O`.	dynamic	`-fno-cse`
`-fdicts-cheap`	Make dictionary-valued expressions seem cheap to the optimiser.	dynamic	`-fno-dicts-cheap`

Flag	Description	Static/Dynamic	Reverse
-fdicts-strict	Make dictionaries strict	dynamic	-fno-dicts-strict
-fdmd-tx-dict-sel	Use a special demand transformer for dictionary selectors. Always enabled by default.	dynamic	-fno-dmd-tx-dict-sel
-fdo-eta-reduction	Enable eta-reduction. Implied by -O.	dynamic	-fno-do-eta-reduction
-fdo-lambda-eta-expansion	Enable lambda eta-expansion. Always enabled by default.	dynamic	-fno-do-lambda-eta-expansion
-feager-blackholing	Turn on eager blackholing	dynamic	-
-fenable-rewrite-rules	Switch on all rewrite rules (including rules generated by automatic specialisation of overloaded functions). Implied by -O.	dynamic	-fno-enable-rewrite-rules
-fexcess-precision	Enable excess intermediate precision	dynamic	-fno-excess-precision
-fexpose-all-unfoldings	Expose all unfoldings, even for very large or recursive functions.	dynamic	-fno-expose-all-unfoldings
-ffloat-in	Turn on the float-in transformation. Implied by -O.	dynamic	-fno-float-in
-ffull-laziness	Turn on full laziness (floating bindings outwards). Implied by -O.	dynamic	-fno-full-laziness
-ffun-to-thunk	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	-fno-fun-to-thunk
-fignore-asserts	Ignore assertions in the source. Implied by -O.	dynamic	-fno-ignore-asserts
-fignore-interface-pragmas	Ignore pragmas in interface files. Implied by -O0 only.	dynamic	-fno-ignore-interface-pragmas
-flate-dmd-anal	Run demand analysis again, at the end of the simplification pipeline	dynamic	-fno-late-dmd-anal
-fliberate-case	Turn on the liberate-case transformation. Implied by -O2.	dynamic	-fno-liberate-case
-fliberate-case-threshold=*n*	Set the size threshold for the liberate-case transformation to *n* (default: 2000)	dynamis	-fno-liberate-case-threshold
-floopification	Turn saturated self-recursive tail-calls into local jumps in the generated assembly. Implied by -O.	dynamic	-fno-loopification

Flag	Description	Static/Dynamic	Reverse
`-fmax-inline-alloc-size=n`	Set the maximum size of inline array allocations to *n* bytes (default: 128). GHC will allocate non-pinned arrays of statically known size in the current nursery block if they're no bigger than *n* bytes, ignoring GC overheap. This value should be quite a bit smaller than the block size (typically: 4096).	dynamic	-
`-fmax-inline-memcpy-insns=n`	Inline memcpy calls if they would generate no more than *n* pseudo instructions (default: 32).	dynamic	-
`-fmax-inline-memset-insns=n`	Inline memset calls if they would generate no more than *n* pseudo instructions (default: 32).	dynamic	-
`-fmax-relevant-binds=n`	Set the maximum number of bindings to display in type error messages (default 6).	dynamic	`-fno-max-relevant-bindings`
`-fmax-simplifier-iterations=n`	Set the max iterations for the simplifier (default 4).	dynamic	-
`-fmax-worker-args=n`	If a worker has that many arguments, none will be unpacked anymore (default: 10)	dynamic	-
`-fno-opt-coercion`	Turn off the coercion optimiser	static	-
`-fno-pre-inlining`	Turn off pre-inlining	dynamic	-
`-fno-state-hack`	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.	static	-
`-fomit-interface-pragmas`	Don't generate interface pragmas. Implied by `-O0` only.	dynamic	`-fno-omit-interface-pragmas`
`-fomit-yields`	Omit heap checks when no allocation is being performed.	dynamic	`-fno-omit-yields`
`-fpedantic-bottoms`	Make GHC be more precise about its treatment of bottom (but see also `-fno-state-hack`). In particular, GHC will not eta-expand through a case expression.	dynamic	`-fno-pedantic-bottoms`

Flag	Description	Static/Dynamic	Reverse
`-fregs-graph`	Use the graph colouring register allocator for register allocation in the native code generator. Implied by `-O2`.	dynamic	`-fno-regs-graph`
`-fregs-iterative`	Use the iterative coalescing graph colouring register allocator in the native code generator.	dynamic	`-fno-regs-iterative`
`-fsimplifier-phases=n`	Set the number of phases for the simplifier (default 2). Ignored with `-O0`.	dynamic	-
`-fsimpl-tick-factor=n`	Set the percentage factor for simplifier ticks (default 100)	dynamic	-
`-fspec-constr`	Turn on the SpecConstr transformation. Implied by `-O2`.	dynamic	`-fno-spec-constr`
`-fspec-constr-count=n`	Set to *n* (default: 3) the maximum number of specialisations that will be created for any one function by the SpecConstr transformation	dynamic	`-fno-spec-constr-count`
`-fspec-constr-threshold=n`	Set the size threshold for the SpecConstr transformation to *n* (default: 2000)	dynamic	`-fno-spec-constr-threshold`
`-fspecialise`	Turn on specialisation of overloaded functions. Implied by `-O`.	dynamic	`-fno-specialise`
`-fstatic-argument-transformation`	Turn on the static argument transformation.	dynamic	`-fno-static-argument-transformation`
`-fstrictness`	Turn on strictness analysis. Implied by `-O`.	dynamic	`-fno-strictness`
`-fstrictness-before=n`	Run an additional strictness analysis before simplifier phase *n*	dynamic	-
`-funbox-small-strict-fields`	Flatten strict constructor fields with a pointer-sized representation. Implied by `-O`.	dynamic	`-fno-unbox-small-strict-fields`
`-funbox-strict-fields`	Flatten strict constructor fields	dynamic	`-fno-unbox-strict-fields`
`-funfolding-creation-threshold=n`	Tweak unfolding settings. Default: 750	dynamic	-
`-funfolding-dict-discount=n`	Tweak unfolding settings. Default: 30	dynamic	-
`-funfolding-fun-discount=n`	Tweak unfolding settings. Default: 60	dynamic	-
`-funfolding-keeness-factor=n`	Tweak unfolding settings. Default: 1.5	dynamic	-
`-funfolding-use-threshold=n`	Tweak unfolding settings. Default: 60	dynamic	-

Flag	Description	Static/Dynamic	Reverse
-fvectorisation-avoidance	Enable vectorisation avoidance. Always enabled by default.	dynamic	-fno-vectorisation-avoidance
-fvectorise	Enable vectorisation of nested data parallelism	dynamic	-fno-vectorise

4.19.16 Profiling options

Chapter 5

Flag	Description	Static/Dynamic	Reverse
-prof	Turn on profiling	dynamic	-
-fprof-auto	Auto-add SCCs to all bindings not marked INLINE	dynamic	-fno-prof-auto
-fprof-auto-top	Auto-add SCCs to all top-level bindings not marked INLINE	dynamic	-fno-prof-auto
-fprof-auto-exported	Auto-add SCCs to all exported bindings not marked INLINE	dynamic	-fno-prof-auto
-fprof-cafs	Auto-add SCCs to all CAFs	dynamic	-fno-prof-cafs
-fno-prof-count-entries	Do not collect entry counts	dynamic	-fprof-count-entries
-ticky	Turn on ticky-ticky profiling	dynamic	-

4.19.17 Program coverage options

Section 5.7

Flag	Description	Static/Dynamic	Reverse
-fhpc	Turn on Haskell program coverage instrumentation	dynamic	-
-hpcdir dir	Directory to deposit .mix files during compilation (default is .hpc)	dynamic	-

4.19.18 Haskell pre-processor options

Section 4.12.4

Flag	Description	Static/Dynamic	Reverse
-F	Enable the use of a pre-processor (set with -pgmF)	dynamic	-

4.19.19 C pre-processor options

Section 4.12.3

Flag	Description	Static/Dynamic	Reverse
`-cpp`	Run the C pre-processor on Haskell source files	dynamic	-
`-Dsymbol[=value]`	Define a symbol in the C pre-processor	dynamic	`-Usymbol`
`-Usymbol`	Undefine a symbol in the C pre-processor	dynamic	-
`-Idir`	Add `dir` to the directory search list for `#include` files	dynamic	-

4.19.20 Code generation options

Section 4.12.5

Flag	Description	Static/Dynamic	Reverse
`-fasm`	Use the native code generator	dynamic	-fllvm
`-fllvm`	Compile using the LLVM code generator	dynamic	-fasm
`-fno-code`	Omit code generation	dynamic	-
`-fwrite-interface`	Always write interface files	dynamic	-
`-fbyte-code`	Generate byte-code	dynamic	-
`-fobject-code`	Generate object code	dynamic	-

4.19.21 Linking options

Section 4.12.6

Flag	Description	Static/Dynamic	Reverse
`-shared`	Generate a shared library (as opposed to an executable)	dynamic	-
`-staticlib`	On Darwin/OS X/iOS only, generate a standalone static library (as opposed to an executable). This is the usual way to compile for iOS.	dynamic	-
`-fPIC`	Generate position-independent code (where available)	dynamic	-
`-dynamic`	Use dynamic Haskell libraries (if available)	dynamic	-
`-dynamic-too`	Build dynamic object files *as well as* static object files during compilation	dynamic	-
`-dyno`	Set the output path for the *dynamically* linked objects	dynamic	-
`-dynosuf`	Set the output suffix for dynamic object files	dynamic	-
`-dynload`	Selects one of a number of modes for finding shared libraries at runtime.	dynamic	-

Flag	Description	Static/Dynamic	Reverse
`-framework` *name*	On Darwin/OS X/iOS only, link in the framework *name*. This option corresponds to the `-framework` option for Apple's Linker.	dynamic	-
`-framework-path` *name*	On Darwin/OS X/iOS only, add *dir* to the list of directories searched for frameworks. This option corresponds to the `-F` option for Apple's Linker.	dynamic	-
`-l`*lib*	Link in library *lib*	dynamic	-
`-L`*dir*	Add *dir* to the list of directories searched for libraries	dynamic	-
`-main-is`	Set main module and function	dynamic	-
`--mk-dll`	DLL-creation mode (Windows only)	dynamic	-
`-no-hs-main`	Don't assume this program contains `main`	dynamic	-
`-rtsopts`, `-rtsopts={none,some,all}`	Control whether the RTS behaviour can be tweaked via command-line flags and the `GHCRTS` environment variable. Using `none` means no RTS flags can be given; `some` means only a minimum of safe options can be given (the default), and `all` (or no argument at all) means that all RTS flags are permitted.	dynamic	-
`-with-rtsopts=`*opts*	Set the default RTS options to *opts*.	dynamic	-
`-no-link`	Omit linking	dynamic	-
`-split-objs`	Split objects (for libraries)	dynamic	-
`-static`	Use static Haskell libraries	dynamic	-
`-threaded`	Use the threaded runtime	dynamic	-
`-debug`	Use the debugging runtime	dynamic	-
`-ticky`	For linking, this simply implies `-debug`; see Section 5.8.	dynamic	-
`-eventlog`	Enable runtime event tracing	dynamic	-
`-fno-gen-manifest`	Do not generate a manifest file (Windows only)	dynamic	-
`-fno-embed-manifest`	Do not embed the manifest in the executable (Windows only)	dynamic	-
`-fno-shared-implib`	Don't generate an import library for a DLL (Windows only)	dynamic	-

Flag	Description	Static/Dynamic	Reverse
`-dylib-install-name` *path*	Set the install name (via `-install_name` passed to Apple's linker), specifying the full install path of the library file. Any libraries or executables that link with it later will pick up that path as their runtime search location for it. (Darwin/OS X only)	dynamic	-
`-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.	dynamic	-

4.19.22 Plugin options

Section 9.3

Flag	Description	Static/Dynamic	Reverse
`-fplugin=`*module*	Load a plugin exported by a given module	dynamic	-
`-fplugin-opt=`*module*:*args*	Give arguments to a plugin module; module must be specified with `-fplugin`	dynamic	-

4.19.23 Replacing phases

Section 4.12.1

Flag	Description	Static/Dynamic	Reverse
`-pgmL` *cmd*	Use *cmd* as the literate pre-processor	dynamic	-
`-pgmP` *cmd*	Use *cmd* as the C pre-processor (with `-cpp` only)	dynamic	-
`-pgmc` *cmd*	Use *cmd* as the C compiler	dynamic	-
`-pgms` *cmd*	Use *cmd* as the splitter	dynamic	-
`-pgma` *cmd*	Use *cmd* as the assembler	dynamic	-
`-pgml` *cmd*	Use *cmd* as the linker	dynamic	-
`-pgmdll` *cmd*	Use *cmd* as the DLL generator	dynamic	-
`-pgmF` *cmd*	Use *cmd* as the pre-processor (with `-F` only)	dynamic	-
`-pgmwindres` *cmd*	Use *cmd* as the program for embedding manifests on Windows.	dynamic	-

Flag	Description	Static/Dynamic	Reverse
`-pgmlibtool` *cmd*	Use *cmd* as the command for libtool (with `-staticlib` only).	dynamic	-

4.19.24 Forcing options to particular phases

Section 4.12.2

Flag	Description	Static/Dynamic	Reverse
`-optL` *option*	pass *option* to the literate pre-processor	dynamic	-
`-optP` *option*	pass *option* to cpp (with `-cpp` only)	dynamic	-
`-optF` *option*	pass *option* to the custom pre-processor	dynamic	-
`-optc` *option*	pass *option* to the C compiler	dynamic	-
`-optlo` *option*	pass *option* to the LLVM optimiser	dynamic	-
`-optlc` *option*	pass *option* to the LLVM compiler	dynamic	-
`-opta` *option*	pass *option* to the assembler	dynamic	-
`-optl` *option*	pass *option* to the linker	dynamic	-
`-optdll` *option*	pass *option* to the DLL generator	dynamic	-
`-optwindres` *option*	pass *option* to `windres`.	dynamic	-

4.19.25 Platform-specific options

Section 4.16

Flag	Description	Static/Dynamic	Reverse
`-msse2`	(x86 only) Use SSE2 for floating point	dynamic	-

4.19.26 Compiler debugging options

Section 4.18

Flag	Description	Static/Dynamic	Reverse
`-dcore-lint`	Turn on internal sanity checking	dynamic	-
`-ddump-to-file`	Dump to files instead of stdout	dynamic	-
`-ddump-asm`	Dump assembly	dynamic	-
`-ddump-bcos`	Dump interpreter byte code	dynamic	-
`-ddump-cmm`	Dump C-- output	dynamic	-
`-ddump-core-stats`	Print a one-line summary of the size of the Core program at the end of the optimisation pipeline	dynamic	-

Flag	Description	Static/Dynamic	Reverse
-ddump-cpranal	Dump output from CPR analysis	dynamic	-
-ddump-cse	Dump CSE output	dynamic	-
-ddump-deriv	Dump deriving output	dynamic	-
-ddump-ds	Dump desugarer output	dynamic	-
-ddump-flatC	Dump "flat" C	dynamic	-
-ddump-foreign	Dump foreign export stubs	dynamic	-
-ddump-hpc	Dump after instrumentation for program coverage	dynamic	-
-ddump-inlinings	Dump inlining info	dynamic	-
-ddump-llvm	Dump LLVM intermediate code	dynamic	-
-ddump-occur-anal	Dump occurrence analysis output	dynamic	-
-ddump-opt-cmm	Dump the results of C-- to C-- optimising passes	dynamic	-
-ddump-parsed	Dump parse tree	dynamic	-
-ddump-prep	Dump prepared core	dynamic	-
-ddump-rn	Dump renamer output	dynamic	-
-ddump-rule-firings	Dump rule firing info	dynamic	-
-ddump-rule-rewrites	Dump detailed rule firing info	dynamic	-
-ddump-rules	Dump rules	dynamic	-
-ddump-vect	Dump vectoriser input and output	dynamic	-
-ddump-simpl	Dump final simplifier output	dynamic	-
-ddump-simpl-iterations	Dump output from each simplifier iteration	dynamic	-
-ddump-spec	Dump specialiser output	dynamic	-
-ddump-splices	Dump TH spliced expressions, and what they evaluate to	dynamic	-
-ddump-stg	Dump final STG	dynamic	-
-ddump-stranal	Dump strictness analyser output	dynamic	-
-ddump-strsigs	Dump strictness signatures	dynamic	-
-ddump-tc	Dump typechecker output	dynamic	-
-ddump-types	Dump type signatures	dynamic	-
-ddump-worker-wrapper	Dump worker-wrapper output	dynamic	-
-ddump-if-trace	Trace interface files	dynamic	-
-ddump-tc-trace	Trace typechecker	dynamic	-
-ddump-vt-trace	Trace vectoriser	dynamic	-
-ddump-rn-trace	Trace renamer	dynamic	-
-ddump-rn-stats	Renamer stats	dynamic	-
-ddump-simpl-stats	Dump simplifier stats	dynamic	-
-dno-debug-output	Suppress unsolicited debugging output	static	-
-dppr-debug	Turn on debug printing (more verbose)	static	-
-dppr-user-length	Set the depth for printing expressions in error msgs	dynamic	-

Flag	Description	Static/Dynamic	Reverse
`-dppr-colsNNN`	Set the width of debugging output. For example `-dppr-cols200`	dynamic	-
`-dppr-case-as-let`	Print single alternative case expressions as strict lets.	dynamic	-
`-dsuppress-all`	In core dumps, suppress everything (except for uniques) that is suppressible.	dynamic	-
`-dsuppress-uniques`	Suppress the printing of uniques in debug output (easier to use **diff**)	dynamic	-
`-dsuppress-idinfo`	Suppress extended information about identifiers where they are bound	dynamic	-
`-dsuppress-module-prefixes`	Suppress the printing of module qualification prefixes	dynamic	-
`-dsuppress-type-signatures`	Suppress type signatures	dynamic	-
`-dsuppress-type-applications`	Suppress type applications	dynamic	-
`-dsuppress-coercions`	Suppress the printing of coercions in Core dumps to make them shorter	dynamic	-
`-dsource-stats`	Dump haskell source stats	dynamic	-
`-dcmm-lint`	C-- pass sanity checking	dynamic	-
`-dstg-lint`	STG pass sanity checking	dynamic	-
`-dstg-stats`	Dump STG stats	dynamic	-
`-dverbose-core2core`	Show output from each core-to-core pass	dynamic	-
`-dverbose-stg2stg`	Show output from each STG-to-STG pass	dynamic	-
`-dshow-passes`	Print out each pass name as it happens	dynamic	-
`-dfaststring-stats`	Show statistics for fast string usage when finished	dynamic	-
`-frule-check`	Report sites with rules that could have fired but didn't. Takes a string argument.	dynamic	-

4.19.27 Misc compiler options

Flag	Description	Static/Dynamic	Reverse
`-jN`	When compiling with `--make`, compile *N* modules in parallel.	dynamic	-
`-fno-hi-version-check`	Don't complain about `.hi` file mismatches	dynamic	-
`-fhistory-size`	Set simplification history size	dynamic	-
`-fno-ghci-history`	Do not use the load/store the GHCi command history from/to `ghci_history`.	dynamic	-

Flag	Description	Static/Dynamic	Reverse
`-fno-ghci-sandbox`	Turn off the GHCi sandbox. Means computations are run in the main thread, rather than a forked thread.	dynamic	-

Chapter 5

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` option, and probably one of the options for adding automatic annotations: `-fprof-auto` 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`, 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.

5.1 Cost centres and cost-centre stacks

GHC's profiling system assigns *costs* to *cost centres*. A cost is simply the time or space (memory) required to evaluate an expression. Cost centres are program annotations around expressions; all costs incurred by the annotated expression are assigned to the enclosing cost centre. Furthermore, GHC will remember the stack of enclosing cost centres for any given expression at run-time and generate a call-tree of cost attributions.

Let's take a look at an example:

```
main = print (fib 30)
fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

Compile and run this program as follows:

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

When a GHC-compiled program is run with the `-p` RTS option, it generates a file called *prog*`.prof`. In this case, the file will contain something like this:

[1] `-fprof-auto` was known as `-auto-all` prior to GHC 7.4.1.

```
        Wed Oct 12 16:14 2011 Time and Allocation Profiling Report  (Final)

           Main +RTS -p -RTS

        total time  =        0.68 secs   (34 ticks @ 20 ms)
        total alloc = 204,677,844 bytes  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc

fib         Main    100.0  100.0


                                                     individual     inherited
COST CENTRE MODULE                  no.     entries  %time %alloc   %time %alloc

MAIN        MAIN                    102           0    0.0    0.0   100.0  100.0
 CAF        GHC.IO.Handle.FD        128           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding.Iconv   120           0    0.0    0.0     0.0    0.0
 CAF        GHC.Conc.Signal         110           0    0.0    0.0     0.0    0.0
 CAF        Main                    108           0    0.0    0.0   100.0  100.0
  main      Main                    204           1    0.0    0.0   100.0  100.0
   fib      Main                    205     2692537  100.0  100.0   100.0  100.0
```

The first part of the file gives the program name and options, and the total time and total memory allocation measured during the run of the program (note that the total memory allocation figure isn't the same as the amount of *live* memory needed by the program at any one time; the latter can be determined using heap profiling, which we will describe later in Section 5.4).

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:

entries The number of times this particular point in the call tree was entered.

individual %time The percentage of the total run time of the program spent at this point in the call tree.

individual %alloc The percentage of the total memory allocations (excluding profiling overheads) of the program made by this call.

inherited %time The percentage of the total run time of the program spent below this point in the call tree.

inherited %alloc 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` RTS option to get the following additional information:

`ticks` The raw number of time "ticks" which were attributed to this cost-centre; from this, we get the `%time` figure mentioned above.

`bytes` Number of bytes allocated in the heap while in this cost-centre; again, this is the raw number from which we get the `%alloc` figure mentioned above.

What about recursive functions, and mutually recursive groups of functions? Where are the costs attributed? Well, although GHC does keep information about which groups of functions called each other recursively, this information isn't displayed in the basic time and allocation profile, instead the call-graph is flattened into a tree as follows: a call to a function that occurs elsewhere on the current stack does not push another entry on the stack, instead the costs for this call are aggregated into the caller[2].

5.1.1 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 is

```
{-# SCC "name" #-} <expression>
```

where `"name"` is an arbitrary string, that will become the name of your cost centre as it appears in the profiling output, and `<expression>` is any Haskell expression. An `SCC` annotation extends as far to the right as possible when parsing. (SCC stands for "Set Cost Centre"). The double quotes can be omitted if `name` is a Haskell identifier, for example:

```
{-# SCC my_function #-} <expression>
```

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:

[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.

```
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
```

5.1.2 Rules for attributing costs

While running a program with profiling turned on, GHC maintains a cost-centre stack behind the scenes, and attributes any costs (memory allocation and time) to whatever the current cost-centre stack is at the time the cost is incurred.

The mechanism is simple: whenever the program evaluates an expression with an SCC annotation, `{-#SCC c -#} E`, the cost centre `c` is pushed on the current stack, and the entry count for this stack is incremented by one. The stack also sometimes has to be saved and restored; in particular when the program creates a *thunk* (a lazy suspension), the current cost-centre stack is stored in the thunk, and restored when the thunk is evaluated. In this way, the cost-centre stack is independent of the actual evaluation order used by GHC at runtime.

At a function call, GHC takes the stack stored in the function being called (which for a top-level function will be empty), and *appends* it to the current stack, ignoring any prefix that is identical to a prefix of the current stack.

We mentioned earlier that lazy computations, i.e. thunks, capture the current stack when they are created, and restore this stack when they are evaluated. What about top-level thunks? They are "created" when the program is compiled, so what stack should we give them? The technical name for a top-level thunk is a CAF ("Constant Applicative Form"). GHC assigns every CAF in a module a stack consisting of the single cost centre `M.CAF`, where `M` is the name of the module. It is also possible to give each CAF a different stack, using the option `-fprof-cafs`. This is especially useful when compiling with `-ffull-laziness` (as is default with `-O` 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`) in order to determine what these CAFs correspond to.

5.2 Compiler options for profiling

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

Without a `-prof` option, your `SCC`s 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`. 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 traceStack in the module Debug.Trace, or the -xc RTS flag (Section 4.17.7) 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, -fprof-auto-top, or -fprof-auto-exported options.

-fno-prof-cafs: Disables any previous -fprof-cafs 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).

5.3 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 or -P or -pa: The -p option produces a standard *time profile* report. It is written into the file *program*.prof.

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

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

-V*secs* 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.

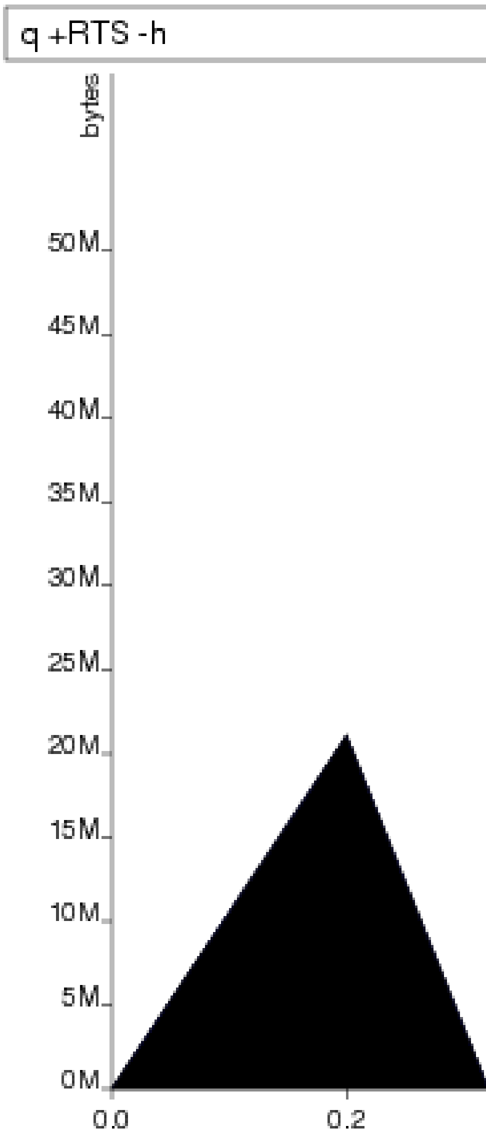
-xc This option causes the runtime to print out the current cost-centre stack whenever an exception is raised. This can be particularly useful for debugging the location of exceptions, such as the notorious Prelude.head:empty list error. See Section 4.17.7.

5.4 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 (Section 5.2).
2. Run it with one of the heap profiling options described below (eg. -h for a basic producer profile). This generates the file *prog*.hp.
3. Run **hp2ps** to produce a Postscript file, *prog*.ps. The **hp2ps** utility is described in detail in Section 5.5.
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 program given above in Section 5.1.1:
You might also want to take a look at hp2any, a more advanced suite of tools (not distributed with GHC) for displaying heap profiles.

5.4.1 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:

-hc (can be shortened to `-h`). Breaks down the graph by the cost-centre stack which produced the data.

-hm Break down the live heap by the module containing the code which produced the data.

-hd 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 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 Break down the graph by *retainer set*. Retainer profiling is described in more detail below (Section 5.4.2).

-hb Break down the graph by *biography*. Biographical profiling is described in more detail below (Section 5.4.3).

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` 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` and `-hb` 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 Section 5.3 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` 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.

5.4.2 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` RTS option:

`-R`*`size`* Restrict the number of elements in a retainer set to *`size`* (default 8).

5.4.2.1 Hints for using retainer profiling

The definition of retainers is designed to reflect a common cause of space leaks: a large structure is retained by an unevaluated computation, and will be released once the computation is forced. A good example is looking up a value in a finite map, where unless the lookup is forced in a timely manner the unevaluated lookup will cause the whole mapping to be retained. These kind of space leaks can often be eliminated by forcing the relevant computations to be performed eagerly, using `seq` or strictness annotations on data constructor fields.

Often a particular data structure is being retained by a chain of unevaluated closures, only the nearest of which will be reported by retainer profiling - for example A retains B, B retains C, and C retains a large structure. There might be a large number of 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.

5.4.3 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.

5.4.4 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` 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 `+RTS -c` option), this is reduced to 2L, and can further be reduced by tweaking the `-F` option. Also add the size of the allocation area (currently a fixed 512Kb).
- The stack isn't counted in the heap profile by default. See the `+RTS -xt` option.
- The program text itself, the C stack, any non-heap data (eg. data allocated by foreign libraries, and data allocated by the RTS), and `mmap()`'d memory are not counted in the heap profile.

5.5 hp2ps--heap profile to PostScript

Usage:

```
hp2ps [flags] [<file>[.hp]]
```

The program **hp2ps** converts a heap profile as 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.

5.5.1 Manipulating the hp file

(Notes kindly offered by Jan-Willem Maessen.)

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

```
JOB "FOO -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.

5.5.2 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.

5.5.3 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 FOO.hp | tail -1 | cut -d : -f 1` FOO.hp \
  | hp2ps > FOO.ps
```

The command **fgrep -n END_SAMPLE FOO.hp** finds the end of every complete sample in `FOO.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 `FOO.hp`. This results in a properly-formatted .hp file which we feed directly to **hp2ps**.

5.5.4 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 -seascape FOO.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 `FOO.ps` the view will update automatically.

This can all be encapsulated in a little script:

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

Occasionally **gv** will choke as it tries to read an incomplete copy of `FOO.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 FOO.hp | tail -1 | cut -d : -f 1` FOO.hp \
  | hp2ps > FOO.ps
gv FOO.ps &
gvpsnum=$!
while [ 1 ] ; do
  sleep 10
  head -`fgrep -n END_SAMPLE FOO.hp | tail -1 | cut -d : -f 1` FOO.hp \
    | hp2ps > FOO.ps
  kill -HUP $gvpsnum
done
```

5.6 Profiling Parallel and Concurrent Programs

Combining `-threaded` and `-prof` is perfectly fine, and indeed it is possible to profile a program running on multiple processors with the `+RTS -N` option.[3]

[3] This feature was added in GHC 7.4.1.

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` 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.

5.7 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.

5.7.1 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 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.

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.h

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

5.7.3 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.

5.7.3.1 hpc report

`hpc report` gives a textual report of coverage. By default, all modules and packages are considered in generating report, unless include or exclude are used. The report is a summary unless the `--per-module` flag is used. The `--xml-output` option allows for tools to use hpc to glean coverage.

```
$ hpc help report
Usage: hpc report [OPTION] .. <TIX_FILE> [<MODULE> [<MODULE> ..]]

Options:

    --per-module                  show module level detail
    --decl-list                   show unused decls
    --exclude=[PACKAGE:][MODULE]  exclude MODULE and/or PACKAGE
    --include=[PACKAGE:][MODULE]  include MODULE and/or PACKAGE
    --srcdir=DIR                  path to source directory of .hs files
                                  multi-use of srcdir possible
    --hpcdir=DIR                  append sub-directory that contains .mix files
                                  default .hpc [rarely used]
    --reset-hpcdirs               empty the list of hpcdir's
                                  [rarely used]
    --xml-output                  show output in XML
```

5.7.3.2 hpc markup

`hpc markup` marks up source files into colored html.

```
$ hpc help markup
Usage: hpc markup [OPTION] .. <TIX_FILE> [<MODULE> [<MODULE> ..]]

Options:
```

```
    --exclude=[PACKAGE:][MODULE]  exclude MODULE and/or PACKAGE
    --include=[PACKAGE:][MODULE]  include MODULE and/or PACKAGE
    --srcdir=DIR                  path to source directory of .hs files
                                  multi-use of srcdir possible
    --hpcdir=DIR                  append sub-directory that contains .mix files
                                  default .hpc [rarely used]
    --reset-hpcdirs               empty the list of hpcdir's
                                  [rarely used]
    --fun-entry-count             show top-level function entry counts
    --highlight-covered           highlight covered code, rather that code gaps
    --destdir=DIR                 path to write output to
```

5.7.3.3 hpc sum

`hpc sum` adds together any number of `.tix` files into a single `.tix` file. `hpc sum` does not change the original `.tix` file; it generates a new `.tix` file.

```
$ hpc help sum
Usage: hpc sum [OPTION] .. <TIX_FILE> [<TIX_FILE> [<TIX_FILE> ..]]
Sum multiple .tix files in a single .tix file

Options:

    --exclude=[PACKAGE:][MODULE]  exclude MODULE and/or PACKAGE
    --include=[PACKAGE:][MODULE]  include MODULE and/or PACKAGE
    --output=FILE                 output FILE
    --union                       use the union of the module namespace (default is ←
       intersection)
```

5.7.3.4 hpc combine

`hpc combine` is the swiss army knife of `hpc`. It can be used to take the difference between `.tix` files, to subtract one `.tix` file from another, or to add two `.tix` files. hpc combine does not change the original `.tix` file; it generates a new `.tix` file.

```
$ hpc help combine
Usage: hpc combine [OPTION] .. <TIX_FILE> <TIX_FILE>
Combine two .tix files in a single .tix file

Options:

    --exclude=[PACKAGE:][MODULE]  exclude MODULE and/or PACKAGE
    --include=[PACKAGE:][MODULE]  include MODULE and/or PACKAGE
    --output=FILE                 output FILE
    --function=FUNCTION           combine .tix files with join function, default = ADD
                                  FUNCTION = ADD | DIFF | SUB
    --union                       use the union of the module namespace (default is ←
       intersection)
```

5.7.3.5 hpc map

hpc map inverts or zeros a `.tix` file. hpc map does not change the original `.tix` file; it generates a new `.tix` file.

```
$ hpc help map
Usage: hpc map [OPTION] .. <TIX_FILE>
Map a function over a single .tix file

Options:
```

```
    --exclude=[PACKAGE:][MODULE]  exclude MODULE and/or PACKAGE
    --include=[PACKAGE:][MODULE]  include MODULE and/or PACKAGE
    --output=FILE                 output FILE
    --function=FUNCTION           apply function to .tix files, default = ID
                                  FUNCTION = ID | INV | ZERO
    --union                       use the union of the module namespace (default is  ↩
        intersection)
```

5.7.3.6 hpc overlay and hpc draft

Overlays are an experimental feature of HPC, a textual description of coverage. hpc draft is used to generate a draft overlay from a .tix file, and hpc overlay generates a .tix files from an overlay.

```
% hpc help overlay
Usage: hpc overlay [OPTION] .. <OVERLAY_FILE> [<OVERLAY_FILE> [...]]

Options:

    --srcdir=DIR   path to source directory of .hs files
                   multi-use of srcdir possible
    --hpcdir=DIR                  append sub-directory that contains .mix files
                                  default .hpc [rarely used]
    --reset-hpcdirs               empty the list of hpcdir's
                                  [rarely used]
    --output=FILE  output FILE
% hpc help draft
Usage: hpc draft [OPTION] .. <TIX_FILE>

Options:

    --exclude=[PACKAGE:][MODULE]  exclude MODULE and/or PACKAGE
    --include=[PACKAGE:][MODULE]  include MODULE and/or PACKAGE
    --srcdir=DIR                  path to source directory of .hs files
                                  multi-use of srcdir possible
    --hpcdir=DIR                  append sub-directory that contains .mix files
                                  default .hpc [rarely used]
    --reset-hpcdirs               empty the list of hpcdir's
                                  [rarely used]
    --output=FILE                 output FILE
```

5.7.4 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. There is no way to change the name of the `.tix` file generated, apart from renaming the binary. HPC does not work with GHCi.

5.8 Using "ticky-ticky" profiling (for implementors)

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 includeds a link to the ticky-ticky profiling page.

Chapter 6

Advice on: sooner, faster, smaller, thriftier

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

6.1 Sooner: producing a program more quickly

Don't use `-O` or (especially) `-O2`: 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`!

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>` option. Increasing the default allocation area size used by the compiler's RTS might also help: use the `+RTS -A<size> -RTS` option.

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.

6.2 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 Chapter 5. 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 can sometimes do a far better job at producing fast code than the native code generator. 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` and `-optlc` 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 `-fwarn-missing-signatures` 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 Section 7.22.9 and Section 7.22.10.

"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` option (see Section 4.7.7).

```
% 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`; next to the signature for any binder, it will print the self-same pragmatic information as would be put in an interface file. (Besides, Core syntax is fun to look at!)

Force key functions to be `INLINE`d (esp. monads): Placing `INLINE` pragmas on certain functions that are used a lot can have a dramatic effect. See Section 7.22.6.1.

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` (don't forget the `-O`).

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` (see Section 4.10.2), 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` is only really a good idea in conjunction with `-O`, because otherwise the extra packing and unpacking won't be optimised away. In fact, it is possible that `-funbox-strict-fields` may worsen performance even *with* `-O`, 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 Section 7.2.1 for some information about using unboxed types.

Before resorting to explicit unboxed types, try using strict constructor fields and `-funbox-strict-fields` 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.

Chapter 8 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` RTS option) indicate that it's doing lots of garbage-collection (say, more than 20% of execution time), more memory might help—with the `-M<size>` or `-A<size>` RTS options (see Section 4.17.3).

6.3 Smaller: producing a program that is smaller

Decrease the "go-for-it" threshold for unfolding smallish expressions. Give a `-funfolding-use-threshold0` 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 `Read`.

Use `strip` on your executables.

6.4 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` RTS option; so... **./a.out +RTS -S -G1**...]

Once again, the profiling facilities (Chapter 5) 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).

Chapter 7

GHC Language Features

As with all known Haskell systems, GHC implements some extensions to the 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.

7.1 Language options

The language option flags control what variation of the language are permitted.

Language options can be controlled in two ways:

- Every language option can 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 Section 7.22.1).

The flag `-fglasgow-exts` is equivalent to enabling the following extensions: `-XConstrainedClassMethods`, `-XDeriveDataTypeable`, `-XDeriveFoldable`, `-XDeriveFunctor`, `-XDeriveGeneric`, `-XDeriveTraversable`, `-XEmptyDataDecls`, `-XExistentialQuantification`, `-XExplicitNamespaces`, `-XFlexibleContexts`, `-XFlexibleInstances`, `-XForeignFunctionInterface`, `-XFunctionalDependencies`, `-XGeneralizedNewtypeDeriving`, `-XImplicitParams`, `-XKindSignatures`, `-XLiberalTypeSynonyms`, `-XMagicHash`, `-XMultiParamTypeClasses`, `-XParallelListComp`, `-XPatternGuards`, `-XPostfixOperators`, `-XRankNTypes`, `-XRecursiveDo`, `-XScopedTypeVariables`, `-XStandaloneDeriving`, `-XTypeOperators`, `-XTypeSynonymInstances`, `-XUnboxedTuples`, `-XUnicodeSyntax`, `-XUnliftedFFITypes`. 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.

7.2 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. (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 `-XMagicHash` extension (Section 7.3.2).

The primops make extensive use of unboxed types and unboxed tuples, which we briefly summarise here.

7.2.1 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. We use the convention (but it is only a convention) that primitive types, values, and operations have a `#` suffix (see Section 7.3.2). For some primitive types we have special syntax for literals, also described in the same section.

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. 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.

There are some restrictions on the use of primitive types:

- The main restriction is that you can't pass a primitive value to a polymorphic function or store one in a polymorphic data type. This rules out things like `[Int#]` (i.e. lists of primitive integers). The reason for this restriction is that polymorphic arguments and constructor fields are assumed to be pointers: if an unboxed integer is stored in one of these, the garbage collector would attempt to follow it, leading to unpredictable space leaks. Or a `seq` operation on the polymorphic component may attempt to dereference the pointer, with disastrous results. Even worse, the unboxed value might be larger than a pointer (`Double#` for instance).
- 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. 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#`.

7.2.2 Unboxed tuples

Unboxed tuples aren't really exported by `GHC.Exts`; they are a syntactic extension enabled by the language flag `-XUnboxedTuples`. An unboxed tuple looks like this:

```
(# e_1, ..., e_n #)
```

where `e_1..e_n` are expressions of any type (primitive or non-primitive). The type of an unboxed tuple looks the same.

Note that when unboxed tuples are enabled, `(#` is a single lexeme, so for example when using operators like `#` and `#-` you need to write `( #)` and `( #-)` rather than `(#)` and `(#-)`.

Unboxed tuples are used for functions that need to return multiple values, but they avoid the heap allocation normally associated with using fully-fledged tuples. When an unboxed tuple is returned, the components are put directly into registers or on the stack; the unboxed tuple itself does not have a composite representation. Many of the primitive operations listed in `primops.txt.pp` return unboxed tuples. In particular, the `IO` and `ST` monads use unboxed tuples to avoid unnecessary allocation during sequences of operations.

There are some restrictions on the use of unboxed tuples:

- Values of unboxed tuple types are subject to the same restrictions as other unboxed types; i.e. they may not be stored in polymorphic data structures or passed to polymorphic functions.
- 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.

7.3 Syntactic extensions

7.3.1 Unicode syntax

The language extension `-XUnicodeSyntax` enables Unicode characters to be used to stand for certain ASCII character sequences. The following alternatives are provided:

ASCII	**Unicode alternative**
`::`	::

ASCII	Unicode alternative
`=>`	⇒
`forall`	∀
`->`	→
`<-`	←
-<	⤙
>-	⤚
-<<	
>>-	
*	★

7.3.2 The magic hash

The language extension `-XMagicHash` 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 `-XMagicHash` extension bring anything into scope. For example, to bring `Int#` into scope you must import `GHC.Prim` (see Section 7.2); the `-XMagicHash` 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 `-XMagicHash` also enables some new forms of literals (see Section 7.2.1):

- `'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#`

7.3.3 Negative literals

The literal `-123` is, according to Haskell98 and Haskell 2010, desugared as `negate (fromInteger 123)`. The language extension `-XNegativeLiterals` 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.

7.3.4 Fractional looking integer literals

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

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

7.3.5 Binary 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 `-XBinaryLiterals` 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 `-XBinaryLiterals` is enabled.

7.3.6 Hierarchical Modules

GHC supports a small extension to the syntax of module names: a module name is allowed to contain a dot `'.'`. This is also known as the "hierarchical module namespace" extension, because it extends the normally flat Haskell module namespace into a more flexible hierarchy of modules.

This extension has very little impact on the language itself; modules names are *always* fully qualified, so you can just think of the fully qualified module name as "the module name". In particular, this means that the full module name must be given after the `module` keyword at the beginning of the module; for example, the module `A.B.C` must begin

```
module A.B.C
```

It is a common strategy to use the `as` keyword to save some typing when using qualified names with hierarchical modules. For example:

```
import qualified Control.Monad.ST.Strict as ST
```

For details on how GHC searches for source and interface files in the presence of hierarchical modules, see Section 4.7.3.

GHC comes with a large collection of libraries arranged hierarchically; see the accompanying library documentation. More libraries to install are available from HackageDB.

7.3.7 Pattern guards

The discussion that follows is an abbreviated version of Simon Peyton Jones's original proposal. (Note that the proposal was written before pattern guards were implemented, so refers to them as unimplemented.)

Suppose we have an abstract data type of finite maps, with a lookup operation:

```
lookup :: FiniteMap -> Int -> Maybe Int
```

The lookup returns `Nothing` if the supplied key is not in the domain of the mapping, and `(Just v)` otherwise, where `v` is the value that the key maps to. Now consider the following definition:

```
clunky env var1 var2 | ok1 && ok2 = val1 + val2
| otherwise  = var1 + var2
where
  m1 = lookup env var1
  m2 = lookup env var2
  ok1 = maybeToBool m1
  ok2 = maybeToBool m2
  val1 = expectJust m1
  val2 = expectJust m2
```

The auxiliary functions are

```
maybeToBool :: Maybe a -> Bool
maybeToBool (Just x) = True
maybeToBool Nothing  = False

expectJust :: Maybe a -> a
expectJust (Just x) = x
expectJust Nothing  = error "Unexpected Nothing"
```

What is `clunky` doing? The guard `ok1 && ok2` checks that both lookups succeed, using `maybeToBool` to convert the `Maybe` types to booleans. The (lazily evaluated) `expectJust` calls extract the values from the results of the lookups, and binds the returned values to `val1` and `val2` respectively. If either lookup fails, then clunky takes the `otherwise` case and returns the sum of its arguments.

This is certainly legal Haskell, but it is a tremendously verbose and un-obvious way to achieve the desired effect. Arguably, a more direct way to write clunky would be to use case expressions:

```
clunky env var1 var2 = case lookup env var1 of
  Nothing -> fail
  Just val1 -> case lookup env var2 of
    Nothing -> fail
    Just val2 -> val1 + val2
where
  fail = var1 + var2
```

This is a bit shorter, but hardly better. Of course, we can rewrite any set of pattern-matching, guarded equations as case expressions; that is precisely what the compiler does when compiling equations! The reason that Haskell provides guarded equations is because they allow us to write down the cases we want to consider, one at a time, independently of each other. This structure is hidden in the case version. Two of the right-hand sides are really the same (`fail`), and the whole expression tends to become more and more indented.

Here is how I would write clunky:

```
clunky env var1 var2
  | Just val1 <- lookup env var1
  , Just val2 <- lookup env var2
  = val1 + val2
...other equations for clunky...
```

The semantics should be clear enough. The qualifiers are matched in order. For a <- qualifier, which I call a pattern guard, the right hand side is evaluated and matched against the pattern on the left. If the match fails then the whole guard fails and the next equation is tried. If it succeeds, then the appropriate binding takes place, and the next qualifier is matched, in the augmented environment. Unlike list comprehensions, however, the type of the expression to the right of the <- is the same as the type of the pattern to its left. The bindings introduced by pattern guards scope over all the remaining guard qualifiers, and over the right hand side of the equation.

Just as with list comprehensions, boolean expressions can be freely mixed with among the pattern guards. For example:

```
f x | [y] <- x
    , y > 3
    , Just z <- h y
    = ...
```

Haskell's current guards therefore emerge as a special case, in which the qualifier list has just one element, a boolean expression.

7.3.8 View patterns

View patterns are enabled by the flag `-XViewPatterns`. 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 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 Section 7.3.7 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 *exp* has type *T1* -> *T2* and *pat* matches a *T2*, then the whole view pattern matches a *T1*.

- Matching: To the equations in Section 3.17.3 of the Haskell 98 Report, add the following:

  ```
  case v of { (e -> p) -> e1 ; _ -> e2 }
   =
  case (e v) of { p -> e1 ; _ -> e2 }
  ```

  That is, to match a variable *v* against a pattern (*exp* -> *pat*), evaluate (*exp* *v*) and match the result against *pat*.

- Efficiency: When the same view function is applied in multiple branches of a function definition or a case expression (e.g., in `size` above), GHC makes an attempt to collect these applications into a single nested case expression, so that the view function is only applied once. Pattern compilation in GHC follows the matrix algorithm described in Chapter 4 of The Implementation of Functional Programming Languages. When the top rows of the first column of a matrix are all view patterns with the "same" expression, these patterns are transformed into a single nested case. This includes, for example, adjacent view patterns that line up in a tuple, as in

```
f ((view -> A, p1), p2) = e1
f ((view -> B, p3), p4) = e2
```

The current notion of when two view pattern expressions are "the same" is very restricted: it is not even full syntactic equality. However, it does include variables, literals, applications, and tuples; e.g., two instances of `view ("hi", "there")` will be collected. However, the current implementation does not compare up to alpha-equivalence, so two instances of `(x, view x -> y)` will not be coalesced.

7.3.9 Pattern synonyms

Pattern synonyms are enabled by the flag `-XPatternSynonyms`, 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.

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
```

Note that in this example, the pattern synonyms `Int` and `Arrow` can also be used as expressions (they are *bidirectional*). This is not necessarily the case: *unidirectional* pattern synonyms can also be declared with the following syntax:

```
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. We can give an explicit inversion of a pattern synonym using the following syntax:

```
pattern Head x <- x:xs where
  Head x = [x]
```

The syntax and semantics of pattern synonyms are elaborated in the following subsections. See the Wiki page for more details.

7.3.9.1 Syntax and scoping of pattern synonyms

A pattern synonym declaration can be either unidirectional or bidirectional. The syntax for unidirectional pattern synonyms is:

```
pattern Name args <- pat
```

and the syntax for bidirectional pattern synonyms is:

```
pattern Name args = pat
```

or

```
pattern Name args <- pat where
  Name args = expr
```

Either prefix or infix syntax can be used.

Pattern synonym declarations can only occur in the top level of a module. In particular, they are not allowed as local definitions. Currently, they also don't work in GHCi, but that is a technical restriction that will be lifted in later versions.

The variables in the left-hand side of the definition are bound by the pattern on the right-hand side. For implicitly 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.

7.3.9.2 Import and export of pattern synonyms

The name of the pattern synonym itself is in the same namespace as proper data constructors. In an export or import specification, you must prefix pattern names with the `pattern` keyword, e.g.:

```
module Example (pattern Single) where
pattern Single x = [x]
```

Without the `pattern` prefix, `Single` 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.

7.3.9.3 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 :: CProv => CReq => t1 -> t2 -> ... -> tN -> t
```

where *`CProv`* and *`CReq`* are type contexts, and *`t1`*, *`t2`*, ..., *`tN`* and *`t`* are types.

A pattern synonym of this type can be used in a pattern if the instatiated (monomorphic) type satisfies the constraints of *`CReq`*. In this case, it extends the context available in the right-hand side of the match with *`CProv`*, just like how an existentially-typed data constructor can extend the context.

For example, in the following program:

```
{-# LANGUAGE PatternSynonyms, GADTs #-}
module ShouldCompile where

data T a where
  MkT :: (Show b) => a -> b -> T a

pattern ExNumPat x = MkT 42 x
```

the inferred pattern type of `ExNumPat` is

```
pattern (Show b) => ExNumPat b :: (Num a, Eq a) => T a
```

and so can be used in a function definition like the following:

```
f :: (Num t, Eq t) => T t -> String
f (ExNumPat x) = show x
```

For bidirectional pattern synonyms, uses as expressions have the type

```
(CProv, CReq) => t1 -> t2 -> ... -> tN -> t
```

So in the previous example, `ExNumPat`, when used in an expression, has type

```
ExNumPat :: (Show b, Num a, Eq a) => b -> T t
```

Pattern synonyms can also be given a type signature in the source program, e.g.:

```
-- Inferred type would be 'a -> [a]'
pattern SinglePair :: (a, a) -> [(a, a)]
pattern SinglePair x = [x]
```

7.3.9.4 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
```

7.3.10 n+k patterns

`n+k` pattern support is disabled by default. To enable it, you can use the `-XNPlusKPatterns` flag.

7.3.11 Traditional record syntax

Traditional record syntax, such as `C {f =x}`, is enabled by default. To disable it, you can use the `-XNoTraditionalRecordSyntax` flag.

7.3.12 The 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 `-XRecursiveDo` (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.

7.3.12.1 Recursive binding groups

The flag `-XRecursiveDo` also introduces a new keyword `rec`, which wraps a mutually-recursive group of monadic statements inside a `do` expression, producing a single statement. Similar to a `let` statement inside a `do`, variables bound in the `rec` are visible throughout the `rec` group, and below it. For example, compare

```
    do { a <- getChar              do { a <- getChar
       ; let { r1 = f a r2            ; rec { r1 <- f a r2
       ;     ; r2 = g r1 }            ;     ; r2 <- g r1 }
       ; return (r1 ++ r2) }          ; return (r1 ++ r2) }
```

In both cases, `r1` and `r2` are available both throughout the `let` or `rec` block, and in the statements that follow it. The difference is that `let` is non-monadic, while `rec` is monadic. (In Haskell `let` is really `letrec`, of course.)

The semantics of `rec` is fairly straightforward. Whenever GHC finds a `rec` group, it will compute its set of bound variables, and will introduce an appropriate call to the underlying monadic value-recursion operator `mfix`, belonging to the `MonadFix` class. Here is an example:

```
rec { b <- f a c     ===>    (b,c) <- mfix (\ ~(b,c) -> do { b <- f a c
    ; c <- f b a }                                         ; c <- f b a
                                                           ; return (b,c) })
```

As usual, the meta-variables `b`, `c` etc., can be arbitrary patterns. In general, the statement `rec` *`ss`* is desugared to the statement

```
vs <- mfix (\ ~vs -> do { ss; return vs })
```

where *`vs`* is a tuple of the variables bound by *`ss`*.

Note in particular that the translation for a `rec` block only involves wrapping a call to `mfix`: it performs no other analysis on the bindings. The latter is the task for the `mdo` notation, which is described next.

7.3.12.2 The `mdo` notation

A `rec`-block tells the compiler where precisely the recursive knot should be tied. It turns out that the placement of the recursive knots can be rather delicate: in particular, we would like the knots to be wrapped around as minimal groups as possible. This process is known as *segmentation*, and is described in detail in Secton 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 flag `-XRecursiveDo`, or the `LANGUAGE RecursiveDo` pragma. (The same flag 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.)

7.3.13 Parallel List Comprehensions

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.

7.3.14 Generalised (SQL-Like) List Comprehensions

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: comprehensio except that the syntax we use differs slightly from the paper.

The extension is enabled with the flag -XTransformListComp.

Here is an example:

```
employees = [ ("Simon", "MS", 80)
, ("Erik", "MS", 100)
, ("Phil", "Ed", 40)
, ("Gordon", "Ed", 45)
, ("Paul", "Yale", 60)]

output = [ (the dept, sum salary)
| (name, dept, salary) <- employees
, then group by dept using groupWith
, then sortWith by (sum salary)
, then take 5 ]
```

In this example, the list output would take on the value:

```
[("Yale", 60), ("Ed", 85), ("MS", 180)]
```

There are three new keywords: group, by, and using. (The functions sortWith and groupWith are not keywords; they are ordinary functions that are exported by GHC.Exts.)

There are five new forms of comprehension qualifier, all introduced by the (existing) keyword then:

- ```
  then f
  ```

  This statement requires that f have the type forall a.[a] -> [a]. You can see an example of its use in the motivating example, as this form is used to apply take 5.

- ```
  then f by e
  ```

  This form is similar to the previous one, but allows you to create a function which will be passed as the first argument to f. As a consequence f must have the type forall a.(a -> t) -> [a] -> [a]. As you can see from the type, this function lets f "project out" some information from the elements of the list it is transforming.

  An example is shown in the opening example, where sortWith is supplied with a function that lets it find out the sum salary for any item in the list comprehension it transforms.

- ```
  then group by e using f
  ```

  This is the most general of the grouping-type statements. In this form, f is required to have type `forall a.(a -> t) -> [a] -> [[a]]`. As with the `then f by e` case above, the first argument is a function supplied to f by the compiler which lets it compute e on every element of the list being transformed. However, unlike the non-grouping case, f additionally partitions the list into a number of sublists: this means that at every point after this statement, binders occurring before it in the comprehension refer to *lists* of possible values, not single values. To help understand this, let's look at an example:

  ```
  -- This works similarly to groupWith in GHC.Exts, but doesn't sort its input first
  groupRuns :: Eq b => (a -> b) -> [a] -> [[a]]
  groupRuns f = groupBy (\x y -> f x == f y)

  output = [ (the x, y)
  | x <- ([1..3] ++ [1..2])
  , y <- [4..6]
  , then group by x using groupRuns ]
  ```

  This results in the variable `output` taking on the value below:

  ```
  [(1, [4, 5, 6]), (2, [4, 5, 6]), (3, [4, 5, 6]), (1, [4, 5, 6]), (2, [4, 5, 6])]
  ```

  Note that we have used the `the` function to change the type of x from a list to its original numeric type. The variable y, in contrast, is left unchanged from the list form introduced by the grouping.

- ```
  then group using f
  ```

  With this form of the group statement, f is required to simply have the type `forall a.[a] -> [[a]]`, which will be used to group up the comprehension so far directly. An example of this form is as follows:

  ```
  output = [ x
  | y <- [1..5]
  , x <- "hello"
  , then group using inits]
  ```

  This will yield a list containing every prefix of the word "hello" written out 5 times:

  ```
  ["","h","he","hel","hell","hello","helloh","hellohe","hellohel","hellohell","hellohello"," ↩
      hellohelloh",...]
  ```

7.3.15 Monad comprehensions

Monad comprehensions generalise the list comprehension notation, including parallel comprehensions (Section 7.3.13) and transform comprehensions (Section 7.3.14) 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 `-XTransformListComp`):

```
[ 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 `-XTransformListComp`):

```
[ 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 `-XParallelListComp`):

```
[ (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` extension is enabled. The types and more detailed examples on how to use comprehensions are explained in the previous chapters Section 7.3.14 and Section 7.3.13. 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` backward compatible to built-in, transform and parallel list comprehensions.

More formally, the desugaring is as follows. We write `D[ e | Q]` to mean the desugaring of the monad comprehension `[ e | Q]`:

```
Expressions: e
Declarations: d
Lists of qualifiers: Q,R,S

-- Basic forms
D[ e | ]                 = return e
D[ e | p <- e, Q ]  = e >>= \p -> D[ e | Q ]
D[ e | e, Q ]            = guard e >> \p -> D[ e | Q ]
D[ e | let d, Q ]        = let d in D[ e | Q ]

-- Parallel comprehensions (iterate for multiple parallel branches)
D[ e | (Q | R), S ]      = mzip D[ Qv | Q ] D[ Rv | R ] >>= \(Qv,Rv) -> D[ e | S ]

-- 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.

Monad comprehensions support rebindable syntax (Section 7.3.16). 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.

7.3.16 Rebindable syntax and the implicit Prelude import

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.)

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 `-XRebindableSyntax` flag 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 `fromRational (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` (Section 7.3.12), and parallel array comprehensions, are unaffected.
- Arrow notation (see Section 7.18) 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!

`-XRebindableSyntax` implies `-XNoImplicitPrelude`.

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.

7.3.17 Postfix operators

The `-XPostfixOperators` flag 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.

7.3.18 Tuple sections

The `-XTupleSections` flag enables Python-style 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 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.

7.3.19 Lambda-case

The `-XLambdaCase` flag 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
```

7.3.20 Empty case alternatives

The `-XEmptyCase` flag 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). 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 should be able to compile with `-fwarn-incomplete-patterns`. (Though the pattern match checking is not yet clever enough to do that.) On the other hand (A) looks dangerous, and GHC doesn't check to make sure that, in fact, the function can never get called.

7.3.21 Multi-way if-expressions

With `-XMultiWayIf` flag 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.

7.3.22 Record field disambiguation

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 `-XDisambiguateRecordFields` flag, 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 Section 7.3.23). For example:

```
module Foo where
  import M
  x=True
  ok3 (MkS { x }) = x+1   -- Uses both disambiguation and punning
```

- With `-XDisambiguateRecordFields` 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.)

7.3.23 Record puns

Record puns are enabled by the flag `-XNamedFieldPuns`.

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.)

7.3.24 Record wildcards

Record wildcards are enabled by the flag `-XRecordWildCards`. This flag implies `-XDisambiguateRecordFields`.

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 (Section 7.3.23); 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.
- Record wildcards may *not* be used in record *updates*. For example this is illegal:

  ```
  f r = r { x = 3, .. }
  ```

- 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).
  - In the case of expressions (but not patterns), the variable `f` is in scope unqualified, apart from the binding of the record selector itself.

  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).

- 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)
```

7.3.25 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`C. Or, in recursive `do` expressions (Section 7.3.12), 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 flag is necessary to enable them.

7.3.26 Import and export extensions

7.3.26.1 Hiding things the imported module doesn't export

Technically in Haskell 2010 this is illegal:

```
module A( f ) where
  f = True

module B where
  import A hiding( g )  -- A does not export g
  g = f
```

The `import A hiding( g )` in module `B` is technically an error (Haskell Report, 5.3.1) because `A` does not export `g`. However GHC allows it, in the interests of supporting backward compatibility; for example, a newer version of `A` might export `g`, and you want `B` to work in either case.

The warning `-fwarn-dodgy-imports`, which is off by default but included with `-W`, warns if you hide something that the imported module does not export.

7.3.26.2 Package-qualified imports

With the `-XPackageImports` flag, 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 Section 4.9.4 for an alternative way of disambiguating between module names.

7.3.26.3 Safe imports

With the `-XSafe`, `-XTrustworthy` and `-XUnsafe` 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 Section 7.29

7.3.26.4 Explicit namespaces in import/export

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 (Section 7.4.4) it becomes possible to declare `(++)` as a *type constructor*. In that case, how would you export or import it?

The `-XExplicitNamespaces` 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 `-XExplicitNamespaces` is implied by `-XTypeOperators` and (for some reason) by `-XTypeFamilies`.

In addition, with `-XPatternSynonyms` 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 Section 7.3.9.2).

7.3.27 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: `-XExplicitForAll`, and hence by `-XScopedTypeVariables`, `-XLiberalTypeSynonyms`, `-XRankNTypes`, `-XExistentialQuantification`

`mdo` Stolen by: `-XRecursiveDo`

`foreign` Stolen by: `-XForeignFunctionInterface`

`rec`, `proc`, `-<`, `>-`, `-<<`, `>>-`, and `(|`, `|)` brackets Stolen by: `-XArrows`

`?varid` Stolen by: `-XImplicitParams`

`[|`, `[e|`, `[p|`, `[d|`, `[t|`, `$(`, `$$(`, `[||`, `[e||`, `$varid`, `$$varid` Stolen by: `-XTemplateHaskell`

`[varid|` Stolen by: `-XQuasiQuotes`

***varid*{#}, *char*#, *string*#, *integer*#, *float*#, *float*##** Stolen by: `-XMagicHash`

`(#`, `#)` Stolen by: `-XUnboxedTuples`

varid*!*varid Stolen by: `-XBangPatterns`

`pattern` Stolen by: `-XPatternSynonyms`

7.4 Extensions to data types and type synonyms

7.4.1 Data types with no constructors

With the `-XEmptyDataDecls` flag (or equivalent LANGUAGE pragma), 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 Section 7.13.5).

Such data types have only one value, namely bottom. Nevertheless, they can be useful when defining "phantom types".

7.4.2 Data type contexts

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`.

7.4.3 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 an operator, beginning with a colon; e.g. `:*:`. The lexical syntax is the same as that for data constructors.
- 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; I'm not sure what it should be.)

7.4.4 Type operators

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 `-XTypeOperators` 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 `-XExplicitNamespaces` (which is implied by `-XExplicitTypeOperators`) GHC allows you to specify the latter by preceding it with the keyword `type`, thus:

```
import M( type (+) )
```

See Section 7.3.26.4.

- The fixity of a type operator may be set using the usual fixity declarations but, as in Section 7.4.3, the function and type constructor share a single fixity.

7.4.5 Liberalised type synonyms

Type synonyms are like macros at the type level, but Haskell 98 imposes many rules on individual synonym declarations. With the `-XLiberalTypeSynonyms` 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 `-XUnboxedTuples`, 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 mal-formedness which isn't detected simply by kind checking:

- Type constructor applied to a type involving for-alls (if `XImpredicativeTypes` 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.

7.4.6 Existentially quantified data constructors

The idea of using existential quantification in data type declarations was suggested by Perry, and implemented in Hope+ (Nigel Perry, *The Implementation of Practical Functional Programming Languages*, PhD Thesis, University of London, 1991). It was later formalised by Laufer and Odersky (*Polymorphic type inference and abstract data types*, TOPLAS, 16(5), pp1411-1430, 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.

7.4.6.1 Why existential?

What has this to do with *existential* quantification? Simply that `MkFoo` has the (nearly) isomorphic type

```
MkFoo :: (exists a . (a, a -> Bool)) -> Foo
```

But Haskell programmers can safely think of the ordinary *universally* quantified type given above, thereby avoiding adding a new existential quantification construct.

7.4.6.2 Existentials and type classes

An easy extension is to allow arbitrary contexts before the constructor. For example:

```
data Baz = forall a. Eq a => Baz1 a a
         | forall b. Show b => Baz2 b (b -> b)
```

The two constructors have the types you'd expect:

```
Baz1 :: forall a. Eq a => a -> a -> Baz
Baz2 :: forall b. Show b => b -> (b -> b) -> Baz
```

But when pattern matching on `Baz1` the matched values can be compared for equality, and when pattern matching on `Baz2` the first matched value can be converted to a string (as well as applying the function to it). So this program is legal:

```
  f :: Baz -> String
  f (Baz1 p q) | p == q    = "Yes"
               | otherwise = "No"
  f (Baz2 v fn)            = show (fn v)
```

Operationally, in a dictionary-passing implementation, the constructors `Baz1` and `Baz2` must store the dictionaries for `Eq` and `Show` respectively, and extract it on pattern matching.

7.4.6.3 Record Constructors

GHC allows existentials to be used with records syntax as well. For example:

```
data Counter a = forall self. NewCounter
    { _this    :: self
    , _inc     :: self -> self
    , _display :: self -> IO ()
    , tag      :: a
    }
```

Here `tag` is a public field, with a well-typed selector function `tag ::Counter a -> a`. 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)
```

7.4.6.4 Restrictions

There are several restrictions on the ways in which existentially-quantified constructors can be use.

- 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!

7.4.7 Declaring data types with explicit constructor signatures

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 Section 7.4.8, can only be declared using this form.

Notice that GADT-style syntax generalises existential types (Section 7.4.6). 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 Section 7.4.6.2. 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 (Section 7.4.8).
- 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 Section 7.4.6.

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 (Section 7.4.6.3), 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
```

7.4.8 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 -XGADTs. The -XGADTs flag also sets -XGADTSyntax and -XMonoLocalBinds.

- A GADT can only be declared using GADT-style syntax (Section 7.4.7); 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 Section 7.4.7, 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.

7.5 Extensions to the "deriving" mechanism

7.5.1 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 mechani

7.5.2 Stand-alone deriving declarations

GHC now allows stand-alone `deriving` declarations, enabled by `-XStandaloneDeriving`:

```
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 (Section 4.7.13).
- 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 `-XFlexibleInstances`, Section 7.6.3.3). 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.

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 Section 7.6.3.
- The stand-alone syntax is generalised for newtypes in exactly the same way that ordinary `deriving` clauses are generalised (Section 7.5.5). 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.

7.5.3 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 `-XDeriveGeneric`, 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 Section 7.26.
- With `-XDeriveFunctor`, you can derive instances of the class `Functor`, defined in `GHC.Base`.
- With `-XDeriveDataTypeable`, you can derive instances of the class `Data`, defined in `Data.Data`. See Section 7.5.4 for deriving `Typeable`.
- With `-XDeriveFoldable`, you can derive instances of the class `Foldable`, defined in `Data.Foldable`.
- With `-XDeriveTraversable`, 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 `-XDeriveTraversable` implies `-XDeriveFunctor` and `-XDeriveFoldable`.

You can also use a standalone deriving declaration instead (see Section 7.5.2).

In each case the appropriate class must be in scope before it can be mentioned in the `deriving` clause.

7.5.4 Deriving `Typeable` instances

The class `Typeable` is very special:

- `Typeable` is kind-polymorphic (see Section 7.8).

- Only derived instances of `Typeable` are allowed; i.e. handwritten instances are forbidden. This ensures that the programmer cannot subert the type system by writing bogus instances.
- With `-XDeriveDataTypeable` GHC allows you to derive instances of `Typeable` for data types or newtypes, using a `deriving` clause, or using a standalone deriving declaration (Section 7.5.2).
- With `-XDataKinds`, deriving `Typeable` for a data type (whether via a deriving clause or standalone deriving) also derives `Typeable` for the promoted data constructors (Section 7.9).
- However, using standalone deriving, you can *also* derive a `Typeable` instance for a data family. You may not add a `deriving(Typeable)` clause to a `data instance` declaration; instead you must use a standalone deriving declaration for the data family.
- Using standalone deriving, you can *also* derive a `Typeable` instance for a type class.
- The flag `-XAutoDeriveTypeable` triggers the generation of derived `Typeable` instances for every datatype, data family, and type class declaration in the module it is used, unless a manually-specified one is already provided. This flag implies `-XDeriveDataTypeable`.

7.5.5 Generalised derived instances 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!

7.5.5.1 Generalising the deriving clause

GHC now permits such instances to be derived instead, using the flag `-XGeneralizedNewtypeDeriving`, so one can write

```
newtype Dollars = Dollars Int deriving (Eq,Show,Num)
```

and the implementation uses the *same* `Num` dictionary for `Dollars` as for `Int`. Notionally, the compiler derives an instance declaration of the form

```
instance Num Int => Num Dollars
```

which just adds or removes the `newtype` constructor according to the type.

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`s 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.

7.5.5.2 A more precise specification

A derived instance is derived only for declarations of these forms (after expansion of any type synonyms)

```
newtype T v1..vn                   = MkT (t vk+1..vn) deriving (C t1..tj)
newtype instance T s1..sk vk+1..vn = MkT (t vk+1..vn) deriving (C t1..tj)
```

where

- `v1..vn` are type variables, and `t`, `s1..sk`, `t1..tj` are types.
- The `(C t1..tj)` is a partial applications of the class `C`, where the arity of `C` is exactly `j+1`. That is, `C` lacks exactly one type argument.
- `k` is chosen so that `C t1..tj (T v1...vk)` is well-kinded. (Or, in the case of a `data instance`, so that `C t1..tj (T s1..sk)` is well kinded.)
- The type `t` is an arbitrary type.
- The type variables `vk+1...vn` do not occur in the types `t`, `s1..sk`, or `t1..tj`.
- `C` is not `Read`, `Show`, `Typeable`, or `Data`. These classes should not "look through" the type or its constructor. You can still derive these classes for a newtype, but it happens in the usual way, not via this new mechanism.
- 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 Section 7.27.)

Then the derived instance is of form declaration is:

```
instance C t1..tj t => 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 built-in derivation applies (section 4.3.3. of the Haskell Report). (For the standard classes `Eq`, `Ord`, `Ix`, and `Bounded` it is immaterial whether the standard method is used or the one described here.)

7.5.6 Deriving any other class

With `-XDeriveAnyClass` you can derive any other class. The compiler will simply generate an empty instance. The instance context will be generated according to the same rules used when deriving `Eq`. This is mostly useful in classes whose minimal set is empty, and especially when writing generic functions. In case you try to derive some class on a newtype, and `-XGeneralizedNewtypeDeriving` is also on, `-XDeriveAnyClass` takes precedence.

7.6 Class and instances declarations

7.6.1 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 t (Simon Peyton Jones, Mark Jones, Erik Meijer).

7.6.1.1 Multi-parameter type classes

Multi-parameter type classes are permitted, with flag `-XMultiParamTypeClasses`. For example:

```
class Collection c a where
  union :: c a -> c a -> c a
  ...etc.
```

7.6.1.2 The superclasses of a class declaration

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 flag `-XFlexibleContexts` (Section 7.13.2) 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 OK:

```
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, 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.

7.6.1.3 Class method types

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`, constrains only the class type variable (in this case `a`). GHC lifts this restriction (flag `-XConstrainedClassMethods`).

7.6.1.4 Default method signatures

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 flag `-XDefaultSignatures`. 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]`.

We use default signatures to simplify generic programming in GHC (Section 7.26).

7.6.1.5 Nullary type classes

Nullary (no parameter) type classes are enabled with `-XMultiTypeClasses`; historically, they were enabled with the (now deprecated) `-XNullaryTypeClasses`. 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 generalized 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
```

7.6.2 Functional dependencies

Functional dependencies are implemented as described by Mark Jones in "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, .

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 ...
```

There should be more documentation, but there isn't (yet). Yell if you need it.

7.6.2.1 Rules for functional dependencies

In a class declaration, all of the class type variables must be reachable (in the sense mentioned in Section 7.13.2) 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
```

7.6.2.2 Background on functional dependencies

The following description of the motivation and use of functional dependencies is taken from the Hugs user manual, reproduced here (with minor changes) by kind permission of Mark Jones.

Consider the following class, intended as part of a library for collection types:

```
class Collects e ce where
    empty  :: ce
    insert :: e -> ce -> ce
    member :: e -> ce -> Bool
```

The type variable e used here represents the element type, while ce is the type of the container itself. Within this framework, we might want to define instances of this class for lists or characteristic functions (both of which can be used to represent collections of any equality type), bit sets (which can be used to represent collections of characters), or hash tables (which can be used to represent any collection whose elements have a hash function). Omitting standard implementation details, this would lead to the following declarations:

```
instance Eq e => Collects e [e] where ...
instance Eq e => Collects e (e -> Bool) where ...
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.

7.6.2.2.1 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.

7.6.2.2.2 Adding functional dependencies

To get a more useful version of the Collects class, Hugs 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 generalization of the proposal for `parametric type classes' that was put forward by Chen, Hudak, and Odersky, or as a special case of Mark Jones's later framework for "improvement" of qualified types. The underlying ideas are also discussed in a more theoretical and abstract setting in a manuscript [implparam], where they are identified as one point in a general design space for systems of implicit parameterization.). 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-one mapping between types.

More generally, dependencies take the form `x1 ...xn -> y1 ...ym`, where x1, ..., xn, and y1, ..., yn are type variables with n>0 and m>=0, meaning that the y parameters are uniquely determined by the x parameters. Spaces can be used as separators if more than one variable appears on any single side of a dependency, as in `t -> a b`. Note that a class may be annotated with multiple dependencies using commas as separators, as in the definition of E above. Some dependencies that we can write in this notation are redundant, and will be rejected because they don't serve any useful purpose, and may instead indicate an error in the program. Examples of dependencies like this include `a -> a` , `a -> a a` , `a ->` , etc. There can also be some redundancy if multiple dependencies are given, as in `a->b`, `b->c` , `a->c` , and in which some subset implies the remaining dependencies. Examples like this are not treated as errors. Note that dependencies appear only in class declarations, and not in any other part of the language. In particular, the syntax for instance declarations, class constraints, and types is completely unchanged.

By including dependencies in a class declaration, we provide a mechanism for the programmer to specify each multiple parameter class more precisely. The compiler, on the other hand, is responsible for ensuring that the set of instances that are in scope at any given point in the program is consistent with any declared dependencies. For example, the following pair of instance declarations cannot appear together in the same scope because they violate the dependency for D, even though either one on its own would be acceptable:

```
instance D Bool Int where ...
instance D Bool Char where ...
```

Note also that the following declaration is not allowed, even by itself:

```
instance D [a] b where ...
```

The problem here is that this instance would allow one particular choice of [a] to be associated with more than one choice for b, which contradicts the dependency specified in the definition of D. More generally, this means that, in any instance of the form:

```
instance D t s where ...
```

for some particular types t and s, the only variables that can appear in s are the ones that appear in t, and hence, if the type t is known, then s will be uniquely determined.

The benefit of including dependency information is that it allows us to define more general multiple parameter classes, without ambiguity problems, and with the benefit of more accurate types. To illustrate this, we return to the collection class example, and annotate the original definition of `Collects` with a simple dependency:

```
class Collects e ce | ce -> e where
   empty  :: ce
   insert :: e -> ce -> ce
   member :: e -> ce -> Bool
```

The dependency `ce -> e` here specifies that the type e of elements is uniquely determined by the type of the collection ce. Note that both parameters of Collects are of kind *; 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.

7.6.3 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.

7.6.3.1 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 Section 7.6.3.5 for flags that loosen the instance resolution rules.

7.6.3.2 Relaxed rules for 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 OK 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 `-XTypeSynonymInstances` flag, 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 `-XFlexibleInstances` flag 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.

  The `-XFlexibleInstances` flag implies `-XTypeSynonymInstances`.

7.6.3.3 Relaxed rules for instance contexts

In Haskell 98, the assertions 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 `-XFlexibleContexts` flag relaxes this rule, as well as the corresponding rule for type signatures (see Section 7.13.2). With this flag the context of the instance declaration can each consist of arbitrary (well-kinded) assertions `(C t1 ...tn)` subject only to the following rules:

1. The Paterson Conditions: for each assertion in the context
   (a) No type variable has more occurrences in the assertion than in the head
   (b) The assertion has fewer constructors and variables (taken together and counting repetitions) than the head
2. The Coverage Condition. For each functional dependency, tvs_{left} `->` tvs_{right}, of the class, every type variable in S(tvs_{right}) must appear in S(tvs_{left}), where S is the substitution mapping each type variable in the class declaration to the corresponding type in the instance declaration.

These restrictions ensure that context reduction terminates: each reduction step makes the problem smaller by at least one constructor. Both the Paterson Conditions and the Coverage Condition are lifted if you give the `-XUndecidableInstances` flag (Section 7.6.3.4). 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 OK:

```
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
```

7.6.3.4 Undecidable instances

Sometimes even the rules of Section 7.6.3.3 are too onerous. 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 (Section 7.6.2) are particularly troublesome. It is tempting to introduce type variables in the context that do not appear in the head, something that is excluded by the normal rules. For example:

```
class HasConverter a b | a -> b where
   convert :: a -> b

data Foo a = MkFoo a

instance (HasConverter a b,Show b) => Show (Foo a) where
   show (MkFoo value) = show (convert value)
```

This is dangerous territory, however. Here, for example, is a program that would make the typechecker loop:

```
class D a
class F a b | a->b
instance F [a] [[a]]
instance (D c, F a c) => D [a]   -- 'c' is not mentioned in the head
```

Similarly, it can be tempting to lift the coverage condition:

```
class Mul a b c | a b -> c where
  (.*.) :: a -> b -> c

instance Mul Int Int Int where (.*.) = (*)
instance Mul Int Float Float where x .*. y = fromIntegral x * y
instance Mul a b c => Mul a [b] [c] where x .*. v = map (x.*.) v
```

The third instance declaration does not obey the coverage condition; and indeed the (somewhat strange) definition:

```
f = \ b x y -> if b then x .*. [y] else y
```

makes instance inference go into a loop, because it requires the constraint `(Mul a [b] b)`.

Nevertheless, GHC allows you to experiment with more liberal rules. If you use the experimental flag `-XUndecidableInstances` , both the Paterson Conditions and the Coverage Condition (described in Section 7.6.3.3) are lifted. Termination is 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 `-fcontext-stack=`*N*.

The `-XUndecidableInstances` flag is also used to lift some of the restricitions imposed on type family instances. See Section 7.7.2.6.

7.6.3.5 Overlapping instances

In general, as discussed in Section 7.6.3.1, *GHC requires that it be unambiguous which instance declaration should be used to resolve a type-class constraint*. GHC also provides a way to 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 irespective 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: `-XOverlappingInstances` and `-XIncoherentInstances` . These flags 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 `-XIncoherentInstances`.
- 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 `-XOverlappingInstances`; 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 `-XOverlappingInstances`; or if the instance is incoherent.

Now suppose that, in some client module, we are searching for an instance of the *target constraint* `(C ty1 ..tyn)`. The search works like this.

- Find all instances I that *match* the target constraint; that is, the target constraint is a substitution instance of I. These instance declarations are the *candidates*.
- 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 arbitary 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 Int Char)` then only the first instance matches, and all is well. Similarly with `(C Bool Bool)`. But if we try to solve `(C Int Bool)`, both instances match and an error is reported.

As a more substantial example of the rules in action, consider

```
instance {-# OVERLAPPABLE #-} context1 => C Int b     where ...  -- (A)
instance {-# OVERLAPPABLE #-} context2 => C a   Bool  where ...  -- (B)
instance {-# OVERLAPPABLE #-} context3 => C a   [b]   where ...  -- (C)
instance {-# OVERLAPPING  #-} context4 => C Int [Int] where ...  -- (D)
```

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 `-XIncoherentInstances` 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 add the flag `-XIncoherentInstances` when compiling the module that contians (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 -XFlexibleContexts flag.

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 -XFlexibleInstances 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 `-XIncoherentInstances`. 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.)

7.6.3.6 Instance signatures: type signatures in instance declarations

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 `-XInstanceSigs` 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 `-XScopedTypeVariables` (Section 7.13.8), the `forall b` scopes over the definition of `foo`, and in particular over the type signature for `xs`.

7.6.4 Overloaded string literals

GHC supports *overloaded string literals*. Normally a string literal has type `String`, but with overloaded string literals enabled (with `-XOverloadedStrings`) 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 with 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 `-XOverloadedStrings` 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.

7.6.5 Overloaded lists

GHC supports *overloading of the list notation*. Let us recap the notation for constructing lists. In Haskell, the list notation can be 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 Section 7.3.8.)

7.6.5.1 The `IsList` class

In the above desugarings, the functions `toList`, `fromList` and `fromListN` are all methods of the `IsList` class, which is itself exported from the `GHC.Exts` module. The type class is defined as follows:

```
class IsList l where
  type Item l

  fromList :: [Item l] -> l
  toList   :: l -> [Item l]
```

```
  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
```

7.6.5.2 Rebindable syntax

When desugaring list notation with `-XOverloadedLists` 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 `-XRebindableSyntax`, then GHC instead uses whatever is in scope with the names of `toList`, `fromList` and `fromListN`. That is, these functions are rebindable; c.f. Section 7.3.16.

7.6.5.3 Defaulting

Currently, the `IsList` class is not accompanied with defaulting rules. Although feasible, not much thought has gone into how to specify the meaning of the default declarations like:

```
default ([a])
```

7.6.5.4 Speculation about the future

The current implementation of the `OverloadedLists` extension can be improved by handling the lists that are only populated with literals in a special way. More specifically, the compiler could allocate such lists statically using a compact representation and allow `IsList` instances to take advantage of the compact representation. Equipped with this capability the `OverloadedLists` extension will be in a good position to subsume the `OverloadedStrings` extension (currently, as a special case, string literals benefit from statically allocated compact representation).

7.7 Type families

Indexed type families form an extension to facilitate type-level programming. Type families are a generalisation of *associated data types* ("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) and *associated type synonyms* ("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). Type families themselves are described in the paper "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. 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 flag `-XTypeFamilies`. Additional information on the use of type families in GHC is available on the Haskell wiki page on type families.

7.7.1 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.

7.7.1.1 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 [http://www.haskell.org/ghc/docs/latest/html/users_gu GADT declarations] named arguments are entirely optional, so that we can declare `Array` alternatively with

```
data family Array :: * -> *
```

7.7.1.2 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.

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             -- WRONG: These two equations together...
foo B = 2             -- ...will produce a type error.
```

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 are - 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.)

7.7.1.3 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.

7.7.2 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.

7.7.2.1 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 w.r.t. 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
```

7.7.2.2 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 family instance declarations are only legitimate when an appropriate family declaration is in scope - just like class instances require the class declaration to be visible. Moreover, each instance declaration has to conform to the kind determined by its family declaration, and the number of type parameters in an instance declaration must match the number of type parameters in the family declaration. Finally, the right-hand side of a type instance must be a monotype (i.e., it may not include foralls) and after the expansion of all saturated vanilla type synonyms, no synonyms, except family synonyms may remain.

7.7.2.3 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 an open type family instances.

7.7.2.4 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                            -- OK!
  H Int  = Int
  H Bool = Bool
  H a    = String
type instance H Char = Char       -- WRONG: cannot have instances of closed family

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
```

7.7.2.5 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 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.

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 Section 2.4.4 for the overlap rules in GHCi.

7.7.2.6 Decidability of type synonym instances

In order to guarantee that type inference in the presence of type families 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 Function 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 `-XUndecidableInstances` 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.

7.7.3 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 then can be the class parameters. Each class parameter may only be used at most once per associated type, but some may be omitted and they may be in an order other than in the class head. Hence, the following contrived example is admissible:

```
  class C a b c where
    type T c a x :: *
```

Here `c` and `a` are class parameters, but the type is also indexed on a third parameter `x`.

7.7.3.1 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
  ...
```

Note the following points:

- The type indexes corresponding to class parameters must have precisely the same shape the type given in the instance head. To have the same "shape" means that the two types are identical modulo renaming of type variables. For example:

```
instance Eq (Elem [e]) => Collects [e] where
  -- Choose one of the following alternatives:
  type Elem [e] = e        -- OK
  type Elem [x] = x        -- OK
  type Elem x   = x        -- BAD; shape of 'x' is different to '[e]'
  type Elem [Maybe x] = x -- BAD: shape of '[Maybe x]' is different to '[e]'
```

- An instances 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 instance for an associated type can be omitted in class instances. In that case, unless there is a default instance (see Section 7.7.3.2), 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 `*`). WARNING: this facility may be withdrawn in the future.

7.7.3.2 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 varaibels bound on the left hand side. However, 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
```

7.7.3.3 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.

7.7.3.4 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 clause (at the top level).

7.7.4 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.

7.7.4.1 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`, *but not the data family `D`*. That (annoyingly) means that you cannot selectively import Y selectively, thus "`import Y( D(D1,D2) )`", because Y does not export `D`. Instead you should list the exports explicitly, thus:

  ```
       module Y( D(..) ) where ...
  or   module Y( module Y, D ) where ...
  ```

7.7.4.2 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.

7.7.5 Type families and instance declarations

Type families require us to extend the rules for the form of instance heads, which are given in Section 7.6.3.2. 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 ...
```

7.8 Kind polymorphism

This section describes *kind polymorphism*, and extension enabled by `-XPolyKinds`. It is described in more detail in the paper Giving Haskell a Promotion, which appeared at TLDI 2012.

7.8.1 Overview of kind polymorphism

Currently there is a lot of code duplication in the way Typeable is implemented (Section 7.5.4):

```
class Typeable (t :: *) where
  typeOf :: t -> TypeRep

class Typeable1 (t :: * -> *) where
  typeOf1 :: t a -> TypeRep

class Typeable2 (t :: * -> * -> *) where
  typeOf2 :: t a b -> TypeRep
```

Kind polymorphism (with -XPolyKinds) allows us to merge all these classes into one:

```
data Proxy t = Proxy

class Typeable t where
  typeOf :: Proxy t -> TypeRep

instance Typeable Int  where typeOf _ = TypeRep
instance Typeable []   where typeOf _ = TypeRep
```

Note that the datatype Proxy has kind forall k.k -> * (inferred by GHC), and the new Typeable class has kind forall k.k -> Constraint.

7.8.2 Overview

Generally speaking, with -XPolyKinds, GHC will infer a polymorphic kind for un-decorated declarations, whenever possible. For example:

```
data T m a = MkT (m a)
-- GHC infers kind   T :: forall k. (k -> *) -> k -> *
```

Just as in the world of terms, you can restrict polymorphism using a kind signature (sometimes called a kind annotation) (-XPolyKinds implies -XKindSignatures):

```
data T m (a :: *) = MkT (m a)
-- GHC now infers kind   T :: (* -> *) -> * -> *
```

There is no "forall" for kind variables. Instead, when binding a type variable, you can simply mention a kind variable in a kind annotation for that type-variable binding, thus:

```
data T (m :: k -> *) a = MkT (m a)
-- GHC now infers kind   T :: forall k. (k -> *) -> k -> *
```

The kind "forall" is placed just outside the outermost type-variable binding whose kind annotation mentions the kind variable. For example

```
f1 :: (forall a m. m a -> Int) -> Int
         -- f1 :: forall (k:BOX).
         --       (forall (a:k) (m:k->*). m a -> Int)
         --       -> Int

f2 :: (forall (a::k) m. m a -> Int) -> Int
         -- f2 :: (forall (k:BOX) (a:k) (m:k->*). m a -> Int)
         --       -> Int
```

Here in f1 there is no kind annotation mentioning the polymorphic kind variable, so k is generalised at the top level of the signature for f1, making the signature for f1 is as polymorphic as possible. But in the case of of f2 we give a kind annotation in the forall (a:k) binding, and GHC therefore puts the kind forall right there too.

(Note: These rules are a bit indirect and clumsy. Perhaps GHC should allow explicit kind quantification. But the implicit quantification (e.g. in the declaration for data type T above) is certainly very convenient, and it is not clear what the syntax for explicit quantification should be.)

7.8.3 Polymorphic kind recursion and complete kind signatures

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 kind signature* for `T`. A complete kind signature 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 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 open type or data family declaration *always* has a complete user-specified kind signature; 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 -> *

class C a where                 -- C  :: k -> Constraint
  type AT a b                   -- AT :: k -> * -> *
```

  In the last example, the variable `a` has an implicit kind variable annotation from the class declaration. It keeps its polymorphic kind in the associated type declaration. The variable `b`, however, gets defaulted to `*`.

- A closed type familey has a complete signature when all of its type variables are annotated and a return kind (with a top-level `::`) is supplied.

7.8.4 Kind inference in closed type families

Although all open type families are considered to have a complete user-specified 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.

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
```

7.8.5 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.

7.9 Datatype promotion

This section describes *data type promotion*, an extension to the kind system that complements kind polymorphism. It is enabled by `-XDataKinds`, and described in more detail in the paper Giving Haskell a Promotion, which appeared at TLDI 2012.

7.9.1 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 lifted types (kind `*`), type constructors (eg. kind `* -> * -> *`), and unlifted types (Section 7.2.1). In particular when using advanced type system features, such as type families (Section 7.7) or GADTs (Section 7.4.8), 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 eg. `Vec Int Char` is a well-kinded type, even though this is not what we intend when defining length-indexed vectors.

With `-XDataKinds`, 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.

7.9.2 Overview

With `-XDataKinds`, GHC automatically promotes every suitable datatype to be a kind, and its (value) constructors to be type constructors. The following types

```
data Nat = Ze | Su 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:

```
Nat :: BOX
Ze :: Nat
Su :: Nat -> Nat

List k :: BOX
Nil  :: List k
Cons :: k -> List k -> List k

Pair k1 k2 :: BOX
Pair :: k1 -> k2 -> Pair k1 k2

Sum k1 k2 :: BOX
L :: k1 -> Sum k1 k2
R :: k2 -> Sum k1 k2
```

where `BOX` is the (unique) sort that classifies kinds. Note that `List`, for instance, does not get sort `BOX -> BOX`, because we do not further classify kinds; all kinds have sort `BOX`.

The following restrictions apply to promotion:

- We promote `data` types and `newtypes`, but not type synonyms, or type/data families (Section 7.7).
- 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.

7.9.3 Distinguishing between types and constructors

Since constructors and types share the same namespace, with promotion you can get ambiguous type names:

```
data P          -- 1

data Prom = P   -- 2

type T = P      -- 1 or promoted 2?
```

In these cases, if you want to refer to the promoted constructor, you should prefix its name with a quote:

```
type T1 = P     -- 1

type T2 = 'P    -- promoted 2
```

Note that promoted datatypes give rise to named kinds. Since these can never be ambiguous, we do not allow quotes in kind names.

Just as in the case of Template Haskell (Section 7.17.1), there is no way to quote a data constructor or type constructor whose second character is a single quote.

7.9.4 Promoted list and tuple types

With `-XDataKinds`, 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 = ...
```

(Note: the declaration for `HCons` also requires `-XTypeOperators` because of infix type operator `(:')`.) 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.

7.9.5 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:

```
type family UnEx (k :: BOX) (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.

7.9.6 Promoting type operators

Type operators are *not* promoted to the kind level. Why not? Because `*` is a kind, parsed the way identifiers are. Thus, if a programmer tried to write `Either * Bool`, would it be `Either` applied to `*` and `Bool`? Or would it be `*` applied to `Either` and `Bool`. To avoid this quagmire, we simply forbid promoting type operators to the kind level.

7.10 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 `XDataKinds` 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 Section 7.3.26.4).

Here is an exampe 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")
```

7.10.1 Runtime Values for Type-Level Literals

Sometimes it is useful to access the value-level literal assocaited 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.

7.10.2 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 evalute 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
```

7.11 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.

7.11.1 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 (Section 7.27). It is exported by `Data.Coerce`, which also contains the documentation. More details and discussion can be found in the paper Safe Coercions".

7.12 The `Constraint` kind

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 constraints, e.g. `?x::Int` (with the `-XImplicitParams` flag)
- Equality constraints, e.g. `a ~ Int` (with the `-XTypeFamilies` or `-XGADTs` flag)

With the `-XConstraintKinds` flag, 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 superclasses, but to do so you must use `-XUndecidableInstances` to signal that you don't mind if the type checker fails to terminate.

7.13 Other type system extensions

7.13.1 Explicit universal quantification (forall)

Haskell type signatures are implicitly quantified. When the language option `-XExplicitForAll` 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.

Of course `forall` becomes a keyword; you can't use `forall` as a type variable any more!

7.13.2 The context of a type signature

The `-XFlexibleContexts` flag 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 `-XFlexibleContexts` these type signatures are perfectly OK

```
  g :: Eq [a] => ...
  g :: Ord (T a ()) => ...
```

The flag `-XFlexibleContexts` also lifts the corresponding restriction on class declarations (Section 7.6.1.2) and instance declarations (Section 7.6.3.3).

7.13.3 Ambiguous types and the ambiguity check

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 `-XAllowAmbiguousTypes` 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 ambiguious 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 is 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. So the language extension `-XAllowAmbiguousTypes` allows you to switch off the ambiguity check. But 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
```

A historical note. GHC used to impose some more restrictive and less principled conditions on type signatures. For type type `forall tv1..tvn (c1, ...,cn) => type` GHC used to require (a) that each universally quantified type variable `tvi` must be "reachable" from `type`, and (b) that every constraint `ci` mentions at least one of the universally quantified type variables `tvi`. These ad-hoc restrictions are completely subsumed by the new ambiguity check. *End of historical note.*

7.13.4 Implicit parameters

Implicit parameters are implemented as described in "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. (Most of the following, still rather incomplete, documentation is due to Jeff Lewis.)

Implicit parameter support is enabled with the option `-XImplicitParams`.

A variable is called *dynamically bound* when it is bound by the calling context of a function and *statically bound* when bound by the callee's context. In Haskell, all variables are statically bound. Dynamic binding of variables is a notion that goes back to Lisp, but was later discarded in more modern incarnations, such as Scheme. Dynamic binding can be very confusing in an untyped language, and unfortunately, typed languages, in particular Hindley-Milner typed languages like Haskell, only support static scoping of variables.

However, by a simple extension to the type class system of Haskell, we can support dynamic binding. Basically, we express the use of a dynamically bound variable as a constraint on the type. These constraints lead to types of the form `(?x::t') => t`, which says "this function uses a dynamically-bound variable `?x` of type `t'`". For example, the following expresses the type of a sort function, implicitly parameterized 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 parameterized sort function in terms of an explicitly parameterized `sortBy` function:

```
sortBy :: (a -> a -> Bool) -> [a] -> [a]

sort   :: (?cmp :: a -> a -> Bool) => [a] -> [a]
sort    = sortBy ?cmp
```

7.13.4.1 Implicit-parameter type constraints

Dynamic binding constraints behave just like other type class constraints in that they are automatically propagated. Thus, when a function is used, its implicit parameters are inherited by the function that called it. For example, our `sort` function might be used to pick out the least value in a list:

```
least   :: (?cmp :: a -> a -> Bool) => [a] -> a
least xs = head (sort xs)
```

Without lifting a finger, the `?cmp` parameter is propagated to become a parameter of `least` as well. With explicit parameters, the default is that parameters must always be explicit propagated. With implicit parameters, the default is to always propagate them.

An implicit-parameter type constraint differs from other type class constraints in the following way: All uses of a particular implicit parameter must have the same type. This means that the type of `(?x, ?x)` is `(?x::a) => (a,a)`, and not `(?x::a, ?x::b) => (a, b)`, as would be the case for type class constraints.

You can't have an implicit parameter in the context of a class or instance declaration. For example, both these declarations are illegal:

```
class (?x::Int) => C a where ...
instance (?x::a) => Foo [a] where ...
```

Reason: exactly which implicit parameter you pick up depends on exactly where you invoke a function. But the ``invocation'' of instance declarations is done behind the scenes by the compiler, so it's hard to figure out exactly where it is done. Easiest thing is to outlaw the offending types.

Implicit-parameter constraints do not cause ambiguity. For example, consider:

```
 f :: (?x :: [a]) => Int -> Int
 f n = n + length ?x

 g :: (Read a, Show a) => String -> String
 g s = show (read s)
```

Here, `g` has an ambiguous type, and is rejected, but `f` is fine. The binding for `?x` at `f`'s call site is quite unambiguous, and fixes the type `a`.

7.13.4.2 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 :: [a] -> a
min  = let ?cmp = (<=) in least
```

A group of implicit-parameter bindings may occur anywhere a normal group of Haskell bindings can occur, except at top level. That is, they can occur in a `let` (including in a list comprehension, or do-notation, or pattern guards), or a `where` clause. Note the following points:

- An implicit-parameter binding group must be a collection of simple bindings to implicit-style variables (no function-style bindings, and no type signatures); these bindings are neither polymorphic or recursive.
- You may not mix implicit-parameter bindings with ordinary bindings in a single `let` expression; use two nested `let`s instead. (In the case of `where` you are stuck, since you can't nest `where` clauses.)
- You may put multiple implicit-parameter bindings in a single binding group; but they are *not* treated as a mutually recursive group (as ordinary `let` bindings are). Instead they are treated as a non-recursive group, simultaneously binding all the implicit parameter. The bindings are not nested, and may be re-ordered without changing the meaning of the program. For example, consider:

  ```
  f t = let { ?x = t; ?y = ?x+(1::Int) } in ?x + ?y
  ```

  The use of `?x` in the binding for `?y` does not "see" the binding for `?x`, so the type of `f` is

  ```
  f :: (?x::Int) => Int -> Int
  ```

7.13.4.3 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.

7.13.4.4 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`.

7.13.5 Explicitly-kinded quantification

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 flag `-XKindSignatures`.

This flag 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.

7.13.6 Arbitrary-rank polymorphism

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 `-XRankNTypes` (which implies `-XExplicitForAll`, Section 7.13.1) 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 Section 7.13.8)

The `-XRankNTypes` 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 flag did not seem worth the bother.

The obselete language options `-XPolymorphicComponents` and `-XRank2Types` are synonyms for `-XRankNTypes`. 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 specifciations.)

7.13.6.1 Examples

In a `data` or `newtype` declaration one can quantify the types of the constructor arguments. Here are several examples:

```
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])
```

As of GHC 7.10, this is deprecated. The `-fwarn-context-quantification` flag detects this situation and issues a warning. In GHC 7.12, 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.

7.13.6.2 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 foralls in it.

What does it mean to "provide" an explicit type for x? You can do that by giving a type signature for x directly, using a pattern type signature (Section 7.13.8), 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.

7.13.6.3 Implicit quantification

GHC performs implicit quantification as follows. *At the top 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 does *not* find the innermost 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 the illegal type
g :: forall a. (Ord a => a -> a) -> Int
         -- NOT
g :: (forall a. Ord a => a -> a) -> Int
```

The latter produces an illegal type, which you might think is silly, but at least the rule is simple. If you want the latter type, you can write your for-alls explicitly. Indeed, doing so is strongly advised for rank-2 types.

7.13.7 Impredicative polymorphism

GHC supports *impredicative polymorphism*, enabled with `-XImpredicativeTypes`. This means that you can 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])`.

The technical details of this extension are described in the paper Boxy types: type inference for higher-rank types and impredicativity, which appeared at ICFP 2006.

7.13.8 Lexically scoped type variables

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` (Section 7.13.8.2). 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.

Lexically-scoped type variables are enabled by `-XScopedTypeVariables`. This flag implies `-XRelaxedPolyRec`.

7.13.8.1 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 (Section 7.13.8.2)
- An expression type signature (Section 7.13.8.3)
- A pattern type signature (Section 7.13.8.4)
- Class and instance declarations (Section 7.13.8.5)

In Haskell, a programmer-written type signature is implicitly quantified over its free type variables (Section 4.1.2 of the Haskell Report). Lexically scoped type variables affect this implicit quantification rules as follows: any type variable that is in scope is *not* universally quantified. For example, if type variable `a` is in scope, then

```
(e :: a -> a)     means     (e :: a -> a)
(e :: b -> b)     means     (e :: forall b. b->b)
(e :: a -> b)     means     (e :: forall b. a->b)
```

7.13.8.2 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.

7.13.8.3 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)`.

7.13.8.4 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.

7.13.8.5 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. For example:

```
class C a where
  op :: [a] -> a

  op xs = let ys::[a]
              ys = reverse xs
          in
          head ys
```

7.13.9 Bindings and generalisation

7.13.9.1 Switching off the dreaded Monomorphism Restriction

Haskell's monomorphism restriction (see Section 4.5.5 of the Haskell Report) can be completely switched off by `-XNoMonomorphismRestriction`. Since GHC 7.8.1, the monomorphism restriction is switched off by default in GHCi's interactive options (see Section 2.8.3).

7.13.9.2 Generalised typing of mutually recursive bindings

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. If `-XRelaxedPolyRec` is specified: *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). With `-XRelaxedPolyRec` 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
```

7.13.9.3 Let-generalisation

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 flag `-XMonoLocalBinds` 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 flag `-XMonoLocalBinds` is implied by `-XTypeFamilies` and `-XGADTs`. You can switch it off again with `-XNoMonoLocalBinds` but type inference becomes less predicatable if you do so. (Read the papers!)

7.14 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.

To run and test a piece of code containing holes, use the `-fdefer-typed-holes` flag. This flag defers errors produced by typed holes and converts them into warnings. The result is that typed hole errors are converted into warnings (controlled by `-fwarn-typed-holes`). The result is that a hole will behave like `undefined`, but with the added benefits that it shows a warning at compile time and will show another warning message if it gets evaluated. This behaviour follows that of the `-fdefer-type-errors` option, which implies `-fdefer-typed-holes`. See Section 7.16.

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 = _
```

Multiple typed holes can be used to find common type variables between expressions. For example:

```
sum :: [Int] -> Int
sum xs = foldr _f _z xs
```

Shows:

```
holes.hs:2:15:
    Found hole `_f' with type: Int -> Int -> Int
    In the first argument of `foldr', namely `_'
    In the expression: foldr _a _b _c
    In an equation for `sum': sum x = foldr _a _b _c

holes.hs:2:17:
    Found hole `_z' with type: Int
    In the second argument of `foldr', namely `_'
    In the expression: foldr _a _b _c
    In an equation for `sum': sum x = foldr _a _b _c
```

Unbound identifiers with the same name are never unified, even within the same function, but always printed 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
```

This ensures that an unbound identifier is never reported with a too polymorphic type, like `forall a.a`, when used multiple times for types that can not be unified.

7.15 Partial Type Signatures

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 Section 7.14 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 Section 7.14, 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 `-XPartialTypeSignatures` flag 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 `-fno-warn-partial-type-signatures` flag.

7.15.1 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.

7.15.1.1 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:
    Found hole '_' with type: Bool
    To use the inferred type, enable PartialTypeSignatures
    In the type signature for 'not'': Bool -> _
```

When a wildcard is not instantiated to a monotype, it will be generalised over, i.e. replaced by a fresh type variable (of which the name will often start with `w_`), e.g.

```
foo :: _ -> _
foo x = x
-- Inferred: forall w_. w_ -> w_

filter' :: _
filter' = filter -- has type forall a. (a -> Bool) -> [a] -> [a]
-- Inferred: (a -> Bool) -> [a] -> [a]
```

7.15.1.2 Named Wildcards

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 w_x => w_x -> String

somethingShowable' :: Show _x => _x -> _
somethingShowable' x = show (not x)
-- Inferred type: Bool -> String
```

Besides an extra-constraints wildcard (see Section 7.15.1.3), 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 `w_x`). 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 `-XNamedWildcards` flag should be enabled. The example below demonstrated the effect.

```
foo :: _a -> _a
foo _ = False
```

Compiling this program without enabling -XNamedWildcards produces the following error message complaining about the type variable _a no matching the actual type Bool.

```
Test.hs:5:9:
    Couldn't match expected type '_a' with actual type 'Bool'
      '_a' is a rigid type variable bound by
           the type signature for foo :: _a -> _a at Test.hs:4:8
    Relevant bindings include foo :: _a -> _a (bound at Test.hs:4:1)
    In the expression: False
    In an equation for 'foo': foo _ = False
```

Compiling this program with -XNamedWildcards enabled produces the following error message reporting the inferred type of the named wildcard _a.

```
Test.hs:4:8: Warning:
    Found hole '_a' with type: Bool
    In the type signature for 'foo': _a -> _a
```

7.15.1.3 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:
    Found hole '_' with inferred constraints: (Enum a, Eq a, Show a)
    To use the inferred type, enable PartialTypeSignatures
    In the type signature for '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:
    Found hole '_' with inferred constraints: (Eq a, Show a)
    To use the inferred type, enable PartialTypeSignatures
    In the type signature for '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:
    Found hole '_' with inferred constraints: ()
    To use the inferred type, enable PartialTypeSignatures
    In the type signature for '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.

7.15.2 Where can they occur?

Partial type signatures are allowed for bindings, pattern and expression signatures. In all other contexts, e.g. type class or type family declarations, they are disallowed. 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_
```

7.16 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 HaskellWiki page or the original paper.

7.16.1 Enabling deferring of type errors

The flag `-fdefer-type-errors` controls whether type errors are deferred to runtime. Type errors will still be emitted as warnings, but will not prevent compilation.

This flag implies the `-fdefer-typed-holes` flag, which enables this behaviour for typed holes. Should you so wish, it is possible to enable `-fdefer-type-errors` without enabling `-fdefer-typed-holes`, by explicitly specifying `-fno-defer-typed-holes` on the commandline after the `-fdefer-type-errors` flag.

At runtime, whenever a term containing a type error would need to be evaluated, the error is converted into a runtime exception. 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 type error.

7.16.2 Deferred type errors in GHCi

The flag `-fdefer-type-errors` 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
```

7.17 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).

There is a Wiki page about Template Haskell at http://www.haskell.org/haskellwiki/Template_Haskell, and that is the best place to look for further details. You may also consult the online Haskell library reference material (look for module `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 (Section 7.17.3) 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.

7.17.1 Syntax

Template Haskell has the following new syntactic constructions. You need to use the flag `-XTemplateHaskell` to switch these syntactic extensions on.

- 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`.

- 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 either a pattern context or an expression 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 Section 7.17.5.

- 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.

- 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.

- Outermost pattern splices may bind variables. By "outermost" here, we refer to a pattern splice that occurs outside of any quotation brackets. For example,

```
mkPat :: Bool -> Q Pat
mkPat True  = [p| (x, y) |]
mkPat False = [p| (y, x) |]

-- in another module:
foo :: (Char, String) -> String
foo $(mkPat True) = x : y

bar :: (String, Char) -> String
bar $(mkPat False) = x : y
```

- Nested pattern splices do *not* bind variables. By "nested" here, we refer to a pattern splice occurring within a quotation bracket. Continuing the example from the last bullet:

  ```
  baz :: Bool -> Q Exp
  baz b = [| quux $(mkPat b) = x + y |]
  ```

  would fail with `x` and `y` being out of scope.

  The difference in treatment of outermost and nested pattern splices is because outermost splices are run at compile time. GHC can then use the result of running the splice when analyzing the expressions within the pattern's scope. Nested splices, on the other hand, are *not* run at compile time; they are run when the bracket is spliced in, sometime later. Since nested pattern splices may refer to local variables, there is no way for GHC to know, at splice compile time, what variables are bound, so it binds none.

- 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 *delcaration 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. 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.

  Concretely, consider the following code

  ```
  module M where
     import ...
     f x = x
     $(th1 4)
     h y = k y y $(blah1)
     $(th2 10)
     w z = $(blah2)
  ```

  In this example

  1. The body of `h` would be unable to refer to the function `w`.

     A `reify` inside the splice `$(th1 ..)` would see the definition of `f`.

  2. A `reify` inside the splice `$(blah1)` would see the definition of `f`, but would not see the definition of `h`.

3. A `reify` inside the splice `$(th2..)` would see the definition of `f`, all the bindings created by `$(th1..)`, and the definition of `h`.
4. A `reify` inside the splice `$(blah2)` would see the same definitions as the splice `$(th2...)`.

(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.)

7.17.2 Using Template Haskell

- The data types and monadic constructor functions for Template Haskell are in the library `Language.Haskell.THSyntax`.
- 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.
- The flag `-ddump-splices` shows the expansion of all top-level splices as they happen.
- If you are building GHC from source, you need at least a stage-2 bootstrap compiler to run Template Haskell. A stage-1 compiler will reject the TH constructs. Reason: TH compiles and runs 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`, `--interactive`, or file-at-a-time). There used to be a restriction to the former two, but that restriction has been lifted.

7.17.3 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 (here we are a Cygwin prompt on Windows):

```
$ ghc --make -XTemplateHaskell main.hs -o main.exe
```

Run "main.exe" and here is your output:

```
$ ./main
Hello
```

7.17.4 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 a workaround. The basic idea is to compile the program twice:

1. Compile the program or library first the normal way, without `-prof`.
2. Then compile it again with `-prof`, 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` flag when building with `-prof` and Template Haskell is used, GHC will emit an error message.

7.17.5 Template Haskell Quasi-quotation

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 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.

The example below shows quasi-quotation in action. The quoter `expr` is bound to a value of type `QuasiQuoter` defined in module `Expr`. The example makes use of an antiquoted variable `n`, indicated by the syntax `'int:n` (this syntax for anti-quotation was defined by the parser's author, *not* by GHC). This binds `n` to the integer value argument of the constructor `IntExpr` when pattern matching. Please see the referenced paper for further details regarding anti-quotation as well as the description of a technique that uses SYB to leverage a single parser of type `String -> a` to generate both an expression parser that returns a value of type `Q Exp` and a pattern parser that returns a value of type `Q Pat`.

Quasiquoters must obey the same stage restrictions as Template Haskell, e.g., in the example, `expr` cannot be defined in `Main.hs` where it is used, but must be imported.

```
{- ------------- file Main.hs --------------- -}
module Main where

import Expr

main :: IO ()
main = do { print $ eval [expr|1 + 2|]
          ; case IntExpr 1 of
              { [expr|'int:n|] -> print n
              ;  _              -> return ()
              }
          }


{- ------------- file Expr.hs --------------- -}
module Expr where
```

```
import qualified Language.Haskell.TH as TH
import Language.Haskell.TH.Quote

data Expr  =  IntExpr Integer
           |  AntiIntExpr String
           |  BinopExpr BinOp Expr Expr
           |  AntiExpr String
    deriving(Show, Typeable, Data)

data BinOp  =  AddOp
            |  SubOp
            |  MulOp
            |  DivOp
    deriving(Show, Typeable, Data)

eval :: Expr -> Integer
eval (IntExpr n)        = n
eval (BinopExpr op x y) = (opToFun op) (eval x) (eval y)
  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
```

7.18 Arrow notation

Arrows are a generalization of monads introduced by John Hughes. For more details, see

- "Generalising Monads to Arrows", John Hughes, in *Science of Computer Programming* 37, pp67–111, 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/`.

With the `-XArrows` flag, GHC supports the arrow notation described in the second of these papers, translating it using combinators from the `Control.Arrow` module. What follows is a brief introduction to the notation; it won't make much sense unless you've read Hughes's paper.

The extension adds a new kind of expression for defining arrows:

```
exp10 ::= ...
       |  proc apat -> cmd
```

where `proc` is a new keyword. The variables of the pattern are bound in the body of the `proc`-expression, which is a new sort of thing called a *command*. The syntax of commands is as follows:

```
cmd   ::= exp10 -<  exp
       |  exp10 -<< exp
       |  cmd0
```

with *cmd*^{0} up to *cmd*^{9} defined using infix operators as for expressions, and

```
cmd10 ::= \ apat ... apat -> cmd
       |  let decls in cmd
       |  if exp then cmd else cmd
       |  case exp of { calts }
       |  do { cstmt ; ... cstmt ; cmd }
       |  fcmd

fcmd  ::= fcmd aexp
       |  ( cmd )
       |  (| aexp cmd ... cmd |)

cstmt ::= let decls
       |  pat <- cmd
       |  rec { cstmt ; ... cstmt [;] }
       |  cmd
```

where `calts` are like `alts` except that the bodies are commands instead of expressions.

Commands produce values, but (like monadic computations) may yield more than one value, or none, and may do other things as well. For the most part, familiarity with monadic notation is a good guide to using commands. However the values of expressions, even monadic ones, are determined by the values of the variables they contain; this is not necessarily the case for commands.

A simple example of the new notation is the expression

```
proc x -> f -< x+1
```

We call this a *procedure* or *arrow abstraction*. As with a lambda expression, the variable `x` is a new variable bound within the `proc`-expression. It refers to the input to the arrow. In the above example, `-<` is not an identifier but an 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.

7.18.1 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 Section 7.23) 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.

7.18.2 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.

7.18.3 Defining your own control structures

As we're seen, arrow notation provides constructs, modelled on those for expressions, for sequencing, value recursion and conditionals. But suitable combinators, which you can define in ordinary Haskell, may also be used to build new commands out of existing ones. The basic idea is that a command defines an arrow from environments to values. These environments assign values to the free local variables of the command. Thus combinators that produce arrows from arrows may also be used to build commands from commands. For example, the `ArrowPlus` class includes a combinator

```
ArrowPlus a => (<+>) :: a b c -> a b c -> a b c
```

so we can use it to build commands:

```
expr' = proc x -> do
                returnA -< x
        <+> do
                symbol Plus -< ()
                y <- term -< ()
                expr' -< x + y
        <+> do
                symbol Minus -< ()
                y <- term -< ()
                expr' -< x - y
```

(The `do` on the first line is needed to prevent the first `<+> ...` from being interpreted as part of the expression on the previous line.) This is equivalent to

```
expr' = (proc x -> returnA -< x)
        <+> (proc x -> do
                symbol Plus -< ()
                y <- term -< ()
```

```
                expr' -< x + y)
        <+> (proc x -> do
                symbol Minus -< ()
                y <- term -< ()
                expr' -< x - y)
```

We are actually using <+> here with the more specific type

```
ArrowPlus a => (<+>) :: a (e,()) c -> a (e,()) c -> a (e,()) c
```

It is essential that this operator be polymorphic in `e` (representing the environment input to the command and thence to its subcommands) and satisfy the corresponding naturality property

```
arr (first k) >>> (f <+> g) = (arr (first k) >>> f) <+> (arr (first k) >>> g)
```

at least for strict `k`. (This should be automatic if you're not using `seq`.) This ensures that environments seen by the subcommands are environments of the whole command, and also allows the translation to safely trim these environments. (The second component of the input pairs can contain unnamed input values, as described in the next section.) The operator must also not use any variable defined within the current arrow abstraction.

We could define our own operator

```
untilA :: ArrowChoice a => a (e,s) () -> a (e,s) Bool -> a (e,s) ()
untilA body cond = proc x ->
        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)|)
```

7.18.4 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
```

7.18.5 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.

7.18.6 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.

7.19 Bang patterns

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 key change is the addition of a new rule to the semantics of pattern matching in the Haskell 98 report. Add new bullet 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`

Bang patterns are enabled by the flag `-XBangPatterns`.

7.19.1 Informal description of bang patterns

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`. 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.

There is one (apparent) exception to this general rule that a bang only makes a difference when it precedes a variable or wild-card: a bang at the top level of a `let` or `where` binding makes the binding strict, regardless of the pattern. (We say "apparent" exception because the Right Way to think of it is that the bang at the top of a binding is not part of the *pattern*; rather it is part of the syntax of the *binding*, creating a "bang-pattern binding".) For example:

```
let ![x,y] = e in b
```

is a bang-pattern binding. Operationally, it behaves just like a case expression:

```
case e of [x,y] -> b
```

Like a case expression, a bang-pattern binding must be non-recursive, and is monomorphic. However, *nested* bangs in a 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`.

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`.

7.19.2 Syntax and semantics

We add a single new production to the syntax of patterns:

```
  pat ::= !pat
```

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
```

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. In the translation box, first apply the following transformation: for each pattern `pi` that is of form `!qi =ei`, transform it to `(xi,!qi) =((),ei)`, and replace `e0` by `(xi `seq` e0)`. Then, when none of the left-hand-side patterns have a bang at the top, apply the rules in the existing box.

The effect of the let rule is to force complete matching of the pattern `qi` before evaluation of the body is begun. The bang is retained in the translated form in case `qi` is a variable, thus:

```
  let !y = f x in b
```

The let-binding can be recursive. However, it is much more common for the let-binding to be non-recursive, in which case the following law holds: `(let !p =rhs in body)` is equivalent to `(case rhs of !p -> body)`

A pattern with a bang at the outermost level is not allowed at the top level of a module.

7.20 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` flag. That is, expressions of the form `assert pred e` will be rewritten to `e`. You can also disable assertions using the `-fignore-asserts` option. The option `-fno-ignore-asserts` 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.

7.21 Static pointers

The language extension `-XStaticPointers` 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.

7.21.1 Using static pointers

Each reference is given a key which can be used to locate it at runtime with `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 `deRefStaticPtr`

The body `e` of a `static e` expression must be a closed expression. That is, there can be no free variables occurring in `e`, i.e. lambda- or let-bound variables bound locally in the context of the expression.

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)
```

While the following definitions are rejected:

```
ref6 = let x = 1 in static x
ref7 y = static (let x = 1 in y)
```

Note that currently, the body `e` in `static e` is restricted to a single identifier when at the GHCi prompt.

7.21.2 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 :: (Typeable a_1, ... , Typeable a_n) => StaticPtr t
```

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 DerivingDataTypeable      #-}
{-# LANGUAGE ExistentialQuantification #-}
{-# LANGUAGE Rank2Types                #-}
{-# LANGUAGE StandaloneDeriving        #-}
{-# LANGUAGE StaticPointers            #-}

import Control.Monad.ST
import GHC.StaticPtr

data Dict c = c => Dict
  deriving Typeable

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))
```

7.22 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 #-}`.

7.22.1 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 Section 7.22).

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 Section 4.5).

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.

7.22.2 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 Section 4.2.2 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 Section 7.22).

7.22.3 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.

7.22.4 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 Section 7.4.3).

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 `-fno-warn-warnings-deprecations`.

7.22.5 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. `-fwarn-missing-methods`, Section 4.8).

This warning can be turned off with the flag `-fno-warn-missing-methods`.

7.22.6 INLINE and NOINLINE pragmas

These pragmas control the inlining of function definitions.

7.22.6.1 INLINE pragma

GHC (with `-O`, 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 Section 4.10.2.

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` (Section 7.22.6.3) and `INLINABLE` (Section 7.22.6.2) pragmas.

7.22.6.2 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` (Section 7.24). 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 Section 7.22.9).
- 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`

7.22.6.3 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).

7.22.6.4 CONLIKE modifier

An INLINE or NOINLINE pragma may have a CONLIKE modifier, which affects matching in RULEs (only). See Section 7.23.4.

7.22.6.5 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:

```
                           -- Before phase 2     Phase 2 and later
  {-# INLINE   [2]  f #-}  --      No                 Yes
  {-# INLINE   [~2] f #-}  --      Yes                No
  {-# NOINLINE [2]  f #-}  --      No                 Maybe
  {-# NOINLINE [~2] f #-}  --      Maybe              No

  {-# INLINE   f #-}       --      Yes                Yes
  {-# NOINLINE f #-}       --      No                 No
```

By "Maybe" we mean that the usual heuristic inlining rules apply (if the function body is small, or it is applied to interesting-looking arguments etc). Another way to understand the semantics is this:

- For both INLINE 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 (Section 7.23).

7.22.7 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.

7.22.8 RULES pragma

The RULES pragma lets you specify rewrite rules. It is described in Section 7.23.

7.22.9 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 (Section 7.22.6.2).
- A `SPECIALIZE` has the effect of generating (a) a specialised version of the function and (b) a rewrite rule (see Section 7.23) 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 (Section 7.22.6.5) to the RULE generated by a SPECIALIZE pragma, just as you can if you write a RULE directly. For example:

  ```
  {-# SPECIALIZE [0] hammeredLookup :: [(Widget, value)] -> Widget -> value #-}
  ```

  generates a specialisation rule that only fires in Phase 0 (the final phase). If you do not specify any phase control in the SPECIALIZE pragma, the phase control is inherited from the inline pragma (if any) of the function. For example:

  ```
  foo :: Num a => a -> a
  foo = ...blah...
  {-# NOINLINE [0] foo #-}
  {-# SPECIALIZE foo :: Int -> Int #-}
  ```

  The NOINLINE pragma tells GHC not to inline foo until Phase 0; and this property is inherited by the specialisation RULE, which will therefore only fire in Phase 0.

  The main reason for using phase control on specialisations is so that you can write optimisation RULES that fire early in the compilation pipeline, and only *then* specialise the calls to the function. If specialisation is done too early, the optimisation rules might fail to fire.

- The type in a SPECIALIZE pragma can be any type that is less polymorphic than the type of the original function. In concrete terms, if the original function is f then the pragma

  ```
  {-# SPECIALIZE f :: <type> #-}
  ```

  is valid if and only if the definition

  ```
  f_spec :: <type>
  f_spec = f
  ```

  is valid. Here are some examples (where we only give the type signature for the original function, not its code):

  ```
  f :: Eq a => a -> b -> b
  {-# SPECIALISE f :: Int -> b -> b #-}

  g :: (Eq a, Ix b) => a -> b -> b
  {-# SPECIALISE g :: (Eq a) => a -> Int -> Int #-}

  h :: Eq a => a -> a -> a
  {-# SPECIALISE h :: (Eq a) => [a] -> [a] -> [a] #-}
  ```

  The last of these examples will generate a RULE with a somewhat-complex left-hand side (try it yourself), so it might not fire very well. If you use this kind of specialisation, let us know how well it works.

7.22.9.1 SPECIALIZE INLINE

A SPECIALIZE pragma can optionally be followed with a INLINE or NOINLINE pragma, optionally followed by a phase, as described in Section 7.22.6. 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 (Section 7.22.6.5) 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.

7.22.9.2 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 Section 7.22.6.2). 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 (with -O) 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.

7.22.9.3 Obsolete SPECIALIZE syntax

Note: 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 Section 7.23.6).

7.22.10 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.

7.22.11 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 reboxed (this is done automatically by the compiler).

Unpacking constructor fields should only be used in conjunction with `-O`[1], in order to expose unfoldings to the compiler so the reboxing can be removed as often as possible. For example:

```
f :: T -> Float
f (T f1 f2) = f1 + f2
```

The compiler will avoid reboxing `f1` and `f2` by inlining `+` on floats, but only when `-O` is on.

Any single-constructor data is eligible for unpacking; for example

```
data T = T {-# UNPACK #-} !(Int,Int)
```

will store the two `Int`s directly in the `T` constructor, by flattening the pair. Multi-level unpacking is also supported:

```
data T = T {-# UNPACK #-} !S
data S = S {-# UNPACK #-} !Int {-# UNPACK #-} !Int
```

will store two unboxed `Int#`s directly in the `T` constructor. The unpacker can see through newtypes, too.

See also the `-funbox-strict-fields` flag, which essentially has the effect of adding `{-# UNPACK #-}` to every strict constructor field.

7.22.12 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` and `-O`, the field of the constructor `T` is not unpacked.

7.22.13 SOURCE pragma

The `{-#SOURCE #-}` pragma is used only in `import` declarations, to break a module loop. It is described in detail in Section 4.7.9.

[1] in fact, UNPACK has no effect without `-O`, for technical reasons (see tick 5252)

7.22.14 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 Section 7.6.3.5. The pragmas are written immediately after the instance keyword, like this:

```
instance {-# OVERLAPPING #-} C t where ...
```

7.23 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 to see what rules fired. If you need more information, then -ddump-rule-firings shows you each individual rule firing and -ddump-rule-rewrites also shows what the code looks like before and after the rewrite.

7.23.1 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 Section 7.22.6.5), 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 Section 9.3), by making it possible to define a RULE that is never run by GHC, but is nevertheless parsed, typechecked etc, so that it is available to the plugin.
- Each 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
```

In `"wrong1"`, the LHS is not an application; in `"wrong2"`, the LHS has a pattern variable in the head.

- 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 `-XScopedTypeVariables` is automatically enabled; see Section 7.13.8.
- Like other pragmas, RULE pragmas are always checked for scope errors, and are typechecked. Typechecking means that the LHS and RHS of a rule are typechecked, and must have the same type. However, rules are only *enabled* if the `-fenable-rewrite-rules` flag is on (see Section 7.23.2).

7.23.2 Semantics

From a semantic point of view:

- Rules are enabled (that is, used during optimisation) by the `-fenable-rewrite-rules` flag. This flag is implied by `-O`, and may be switched off (as usual) by `-fno-enable-rewrite-rules`. (NB: enabling `-fenable-rewrite-rules` without `-O` may not do what you expect, though, because without `-O` GHC ignores all optimisation information in interface files; see `-fignore-interface-pragmas`, Section 4.10.2.) Note that `-fenable-rewrite-rules` 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 Section 7.23.6 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.

7.23.3 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 `-fwarn-inline-rule-shadowing` (see Section 4.8) warns about this situation.

7.23.4 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.

7.23.5 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.

7.23.6 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` *whenever the types match*. What is more, this rule does not need to be in the same file as `genericLookup`, unlike the `SPECIALIZE` pragmas which currently do (so that they have an original definition available to specialise).

It is *Your Responsibility* to make sure that `intLookup` really behaves as a specialised version of `genericLookup`!!!

An example in which using `RULES` for specialisation will Win Big:

```
toDouble :: Real a => a -> Double
toDouble = fromRational . toRational

{-# RULES "toDouble/Int" toDouble = i2d #-}
i2d (I# i) = D# (int2Double# i) -- uses Glasgow prim-op directly
```

The `i2d` function is virtually one machine instruction; the default conversion—via an intermediate `Rational`—is obscenely expensive by comparison.

7.23.7 Controlling what's going on in rewrite rules

- Use `-ddump-rules` 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` to see what rules are being fired. If you add `-dppr-debug` you get a more detailed listing.
- Use `-ddump-rule-firings` or `-ddump-rule-rewrites` to see in great detail what rules are being fired. If you add `-dppr-debug` you get a still more detailed listing.
- The definition of (say) `build` in `GHC/Base.lhs` 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.lhs` 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.lhs`.

7.24 Special built-in functions

GHC has a few built-in functions with special behaviour. In particular:

- `inline` allows control over inlining on a per-call-site basis.
- `lazy` restrains the strictness analyser.
- `oneShot` gives a hint to the compiler about how often a function is being called.

7.25 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, pp94-105. These have been removed and replaced by the more general support for generic programming.

7.26 Generic programming

Using a combination of `-XDeriveGeneric` (Section 7.5.4), `-XDefaultSignatures` (Section 7.6.1.4), and `-XDeriveAnyClass` (Section 7.5.6), 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.

7.26.1 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 where
  type Rep1 f :: * -> *

  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`. Instances of these classes can be derived by GHC with the `-XDeriveGeneric` (Section 7.5.4), 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` gets the following representation:

```
instance Generic (UserTree a) where
  -- Representation type
  type Rep (UserTree a) =
    M1 D D1UserTree (
          M1 C C1_0UserTree (
                M1 S NoSelector (K1 R a)
            :*: M1 S NoSelector (K1 R (UserTree a))
            :*: M1 S NoSelector (K1 R (UserTree a)))
      :+: M1 C C1_1UserTree 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

-- Meta-information
data D1UserTree
data C1_0UserTree
data C1_1UserTree

instance Datatype D1UserTree where
  datatypeName _ = "UserTree"
  moduleName _   = "Main"

instance Constructor C1_0UserTree where
  conName _ = "Node"

instance Constructor C1_1UserTree where
  conName _ = "Leaf"
```

This representation is generated automatically if a `deriving Generic` clause is attached to the datatype. Standalone deriving can also be used.

7.26.2 Writing generic functions

A generic function is defined by creating a class and giving instances for each of the representation types of `GHC.Generics`. As an example we show generic serialization:

```
data Bin = O | I

class GSerialize f where
  gput :: f a -> [Bin]

instance GSerialize U1 where
```

```
  gput U1 = []

instance (GSerialize a, GSerialize b) => GSerialize (a :*: b) where
  gput (x :*: y) = gput x ++ gput y

instance (GSerialize a, GSerialize b) => GSerialize (a :+: b) where
  gput (L1 x) = O : gput x
  gput (R1 x) = I : gput x

instance (GSerialize a) => GSerialize (M1 i c a) where
  gput (M1 x) = gput x

instance (Serialize a) => GSerialize (K1 i a) where
  gput (K1 x) = put x
```

Typically this class will not be exported, as it only makes sense to have instances for the representation types.

7.26.3 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 -> [Bit]
  put = gput . from
```

Here we use a default signature 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 `-XDeriveAnyClass`, 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.

7.26.4 More information

For more details please refer to the HaskellWiki page or the original paper:

- 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.

7.27 Roles

Using `-XGeneralizedNewtypeDeriving` (Section 7.5.5.1), 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 comput published at POPL 2011.

7.27.1 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.

7.27.2 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`.

7.27.3 Role annotations

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 `-XRoleAnnotations`) 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 `-XIncoherentInstances` to allow non-nominal roles for classes.

The other place where role annotations may be necessary are in `hs-boot` files (Section 4.7.9), 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
```

7.28 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.

7.28.1 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.

7.28.2 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.

7.28.3 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`, and run it with the RTS `-N` option; see Section 4.15). The runtime will schedule the running Haskell threads among the available OS threads, running as many in parallel as you specified with the `-N` RTS option.

GHC only supports parallelism on a shared-memory multiprocessor. Glasgow Parallel Haskell (GPH) supports running Parallel Haskell programs on both clusters of machines, and single multiprocessors. GPH is developed and distributed separately from GHC (see The GPH Page). However, the current version of GPH is based on a much older version of GHC (4.06).

7.28.4 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 (Section 7.28.1), 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 Section 4.17.3.

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`.

7.28.5 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.

7.29 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. Safe Haskell is aimed to be as minimal as possible while still providing strong enough guarantees about compiled Haskell code for more advance secure systems to be built on top of it.

While this is the use case that Safe Haskell was motivated by it is important to understand that what Safe Haskell is tracking and enforcing is a stricter form of type safety than is usually guaranteed in Haskell. As part of this, Safe Haskell is run during every compilation of GHC, tracking safety and inferring it even for modules that don't explicitly use Safe Haskell. Please refer to section Section 7.29.5 for more details of this. This also means that there are some design choices that from a security point of view may seem strange but when thought of from the angle of tracking type safety are logical. Feedback on the current design and this tension between the security and type safety view points is welcome.

The design of Safe Haskell covers the following aspects:

- A safe language dialect of Haskell that provides 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 flag. This can be manipulated to either compromise a users system at compilation time, or to modify the source code just before compilation to try to alter set Safe Haskell flags. This is discussed further in section Section 7.29.7.

7.29.1 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

7.29.1.1 Strict type-safety (good style)

Haskell offers a powerful type system and separation of pure and effectual functions through the `IO` monad. There are several loop holes in the type system though, the most obvious offender being the `unsafePerformIO :: IO a -> a` function. The safe language dialect of Safe Haskell disallows the use of such functions. This can be useful for a variety of purposes as it makes Haskell code easier to analyse and reason about. It also codifies an existing culture in the Haskell community of trying to avoid using such 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`.

7.29.1.2 Building secure systems (restricted IO Monads)

Systems such as information flow control security, capability based security systems and DSLs for working with encrypted data.. etc can be built in the Haskell language simply as a library. However they require guarantees about the properties of the Haskell language that aren't true in the general case where uses of functions like `unsafePerformIO` are allowed. Safe Haskell is designed to give users enough guarantees about the safety properties of compiled code so that such secure systems can be built.

As an example lets 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 that only allows a safe subset of `IO` actions to be executed. We define the plugin interface here so that it requires the plugin module, `Danger`, to export a single computation, `Danger.runMe`, of type `RIO ()`, where `RIO` is a new monad defined as follows:

```
-- Either of the following Safe Haskell pragmas would do
{-# LANGUAGE Trustworthy #-}
{-# 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 compile Danger using the new Safe Haskell `-XSafe` flag:

```
{-# LANGUAGE Safe #-}
module Danger ( runMe ) where

runMe :: RIO ()
runMe = ...
```

Before going into the Safe Haskell details, lets point out some of the reasons this design 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 the Danger module 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 the untrusted Danger module 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 this. The only way to stop this would be to patch or remove packages with known vulnerabilities even if they should only be used by trusted code such as the RIO module.

To stop these attacks Safe Haskell can be used. This is done by compiling the RIO module with the `-XTrustworthy` flag and compiling the Danger module with the `-XSafe` flag.

The use of the `-XSafe` flag to compile the Danger module restricts the features of Haskell that can be used to a safe subset. This includes disallowing `unsafePerformIO`, Template Haskell, pure FFI functions, RULES and restricting the operation of Overlapping Instances. The `-XSafe` flag also restricts the modules can be imported by Danger to only those that are considered trusted. Trusted modules are those compiled with `-XSafe`, where GHC provides a mechanical guarantee that the code is safe. Or those modules compiled with `-XTrustworthy`, where the module author claims that the module is Safe.

This is why the RIO module is compiled with `-XTrustworthy`, to allow the Danger module to import it. The `-XTrustworthy` flag doesn't place any restrictions on the module like `-XSafe` does. 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. The use of `-XTrustworthy` by itself marks the module as trusted. There is an issue here as `-XTrustworthy` may be used by an arbitrary module and module author. To control the use of trustworthy modules it is recommended to use the `-fpackage-trust` flag. This flag adds an extra requirement to the trust check for trustworthy modules, such that for trustworthy modules to be considered trusted, and allowed to be used in `-XSafe` 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 `-XSafe` , 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 Section 7.29.4.

In the example, Danger can import module RIO because RIO is marked trustworthy. 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.

7.29.2 Safe Language

The Safe Haskell *safe language* guarantees the following properties:

- *Referential transparency* — Functions in the safe language are deterministic, evaluating them will not cause any side effects. Functions in the `IO` monad are still allowed and behave as usual. Any pure function though, as according to its type, is guaranteed to indeed be pure. This property allows a user of the safe language to trust the types. This means, for example, that the `unsafePerformIO :: IO a -> a` function is disallowed in the safe language.
- *Module boundary control* — Haskell code compiled using the safe language is guaranteed to only access symbols that are publicly available to it through other modules export lists. An important part of this is that safe compiled code is not able to examine or create data values using data constructors that it cannot import. If a module M establishes some invariants through careful use of its export list then code compiled using the safe language that imports M is guaranteed to respect those invariants. Because of this, *Template Haskell* is disabled in the safe language as it can be used to violate this property.
- *Semantic consistency* — 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. In addition, in any module that imports a safe language module, expressions that compile both with and without the safe import have the same meaning in both cases. That is, importing a module using 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 *Overlapping Instances* extension as it can violate this property.

These three 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.

Lets now look at the details of the safe language. In the safe language dialect (enabled by `-XSafe`) we disable completely the following features:

- *TemplateHaskell* — Is particularly dangerous, as it can cause side effects even at compilation time and can be used to access constructors of abstract data types.

In the safe language dialect we restrict the following features:

- *ForeignFunctionInterface* — This is mostly safe, but foreign import declarations that import a function with a non-IO type are disallowed. All FFI imports must reside in the IO Monad.
- *RULES* — As they can change the behaviour of trusted code in unanticipated ways, violating semantic consistency, they are restricted in function. Specifically any RULES defined in a module M compiled with `-XSafe` are dropped. RULES defined in trustworthy modules that M imports are still valid and will fire as usual.
- *OverlappingInstances* — This extension can be used to violate semantic consistency, because malicious code could redefine a type instance (by containing a more specific instance definition) in a way that changes the behaviour of code importing the untrusted module. The extension is not disabled for a module M compiled with `-XSafe` but restricted. While M can define overlapping instance declarations, they can only overlap other instance declaration defined in M. If in a module N that imports M, at a call site that uses a type-class function there is a choice of which instance to use (i.e. an overlap) and the most specific instances is from M, then all the other choices must also be from M. If not, a compilation error will occur. A simple way to think of this is a *same origin policy* for overlapping instances defined in Safe compiled modules.
- *Data.Typeable* — We restrict Typeable instances to only derived ones (offered by GHC through the -XDeriveDataTypeable extension). Hand crafted instances of the Typeable type class are not allowed in Safe Haskell as this can easily be abused to unsafely coerce between types.

7.29.3 Safe Imports

Safe Haskell enables a small extension to the usual import syntax of Haskell, adding a *safe* keyword:

```
impdecl -> import [safe] [qualified] modid [as modid] [impspec]
```

When used, the module being imported with the safe keyword must be a trusted module, otherwise a compilation error will occur. The safe import extension is enabled by either of the `-XSafe`, `-XTrustworthy`, or `-XUnsafe` flags and corresponding PRAGMA's. When the `-XSafe` flag is used, the safe keyword is allowed but meaningless, every import is required to be safe regardless.

7.29.4 Trust and Safe Haskell Modes

The Safe Haskell extension introduces the following three language flags:

- *-XSafe* — 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.
- *-XTrustworthy* — 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 or place any restrictions on the allowed Haskell code. 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.
- *-XUnsafe* — Marks the module being compiled as unsafe so that modules compiled using `-XSafe` can't import it.

The procedure to check if a module is trusted or not depends on if the `-fpackage-trust` flag is present. The check is very similar in both cases with the presence of the `-fpackage-trust` flag simply enabling an extra requirement for trustworthy modules to be regarded as trusted.

7.29.4.1 Trust check (`-fpackage-trust` disabled)

A *module M in a package P is trusted by a client C* if and only if:

- Both of these hold:
  - The module was compiled with `-XSafe`

- All of M's direct imports are trusted by C
- *OR* all of these hold:
  - The module was compiled with `-XTrustworthy`
  - 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 -XTrustworthy and it will be trusted regardless of what it does. To control this there is an additional definition of package trust (enabled with the `-fpackage-trust` flag). The point of package trusts is to require that the client C explicitly say which packages are allowed to contain trustworthy modules. That is, C establishes that it trusts a package P and its author and so trust the modules in P that use `-XTrustworthy`. When package trust is enabled, any modules that are considered trustworthy but reside in a package that isn't trusted are not considered trusted. A more formal definition is given in the next section.

7.29.4.2 Trust check (`-fpackage-trust` enabled)

When the `-fpackage-trust` flag is enabled, whether or not a module is trusted depends on a notion of trust for packages, which is determined by the client C invoking GHC (i.e. you). 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.

When the `-fpackage-trust` 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 `-XSafe`
  - All of M's direct imports are trusted by C
- *OR* all of these hold:
  - The module was compiled with `-XTrustworthy`
  - 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 `-XTrustworthy` compiled modules.

The reason there are two modes of checking trust is that the extra requirement enabled by `-fpackage-trust` 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 users machine. A maintainer of a package `foo` that uses Safe Haskell so that security conscious Haskellers can use `foo` now may have other users of `foo` who don't know or care about Safe Haskell complaining about compilation problems they are having with `foo`because a package `bar`that foo requires, isn't trusted on their machine. In this sense, the `-fpackage-trust` flag can be thought of as a flag to properly turn on Safe Haskell while without it, it's operating in a covert fashion.

Having the `-fpackage-trust` flag also nicely unifies the semantics of how Safe Haskell works when used explicitly and how modules are inferred as safe.

7.29.4.3 Example

```
Package Wuggle:
   {-# LANGUAGE Safe #-}
   module Buggle where
     import Prelude
     f x = ...blah...

Package P:
   {-# LANGUAGE Trustworthy #-}
   module M where
     import System.IO.Unsafe
     import safe Buggle
```

Suppose a client C decides to trust package P. Then does C trust module M? To decide, GHC must check M's imports — M imports System.IO.Unsafe. M was compiled with `-XTrustworthy`, so P's author takes responsibility for that import. C trusts P's author, so C trusts M to only use its unsafe imports in a safe and consistent manner with respect to the API M exposes. M also has a safe import of Buggle, so for this import P's author takes no responsibility for the safety, so GHC must check whether Buggle is trusted by C. Is it? Well, it is compiled with `-XSafe`, so the code in Buggle itself is machine-checked to be OK, but again under the assumption that all of Buggle's imports are trusted by C. Prelude comes from base, which C trusts, and is compiled with `-XTrustworthy` (While Prelude is typically imported implicitly, it still obeys the same rules outlined here). So Buggle is considered trusted.

Notice that C didn't need to trust package Wuggle; the machine checking is enough. C only needs to trust packages that contain `-XTrustworthy` modules.

7.29.4.4 Trustworthy Requirements

Module authors using the `-XTrustworthy` language extension for a module M should ensure that M's public API (the symbols exposed by its export list) can't be used in an unsafe manner. This mean that symbols exported should respect type safety and referential transparency.

7.29.4.5 Package Trust

Safe Haskell gives packages a new Boolean property, that of trust. Several new options are available at the GHC command-line to specify the trust property of packages:

- *-trust P* — Exposes package P if it was hidden and considers it a trusted package regardless of the package database.
- *-distrust P* — Exposes package P 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 Section 4.9.

7.29.5 Safe Haskell Inference

In the case where a module is compiled without one of `-XSafe`, `-XTrustworthy` or `-XUnsafe` 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 `-XSafe` flag, then M is marked as safe. If M would fail to compile under the `-XSafe` flag, then it is marked as unsafe.

When should you use Safe Haskell inference and when should you use an explicit `-XSafe` flag? The later case should be used when you have a hard requirement that the module be safe. That is, the use cases outlined and the purpose for which Safe Haskell is intended: compiling untrusted code. Safe inference is meant to be used by ordinary Haskell programmers. Users who probably don't care about Safe Haskell.

Say you are writing a Haskell library. Then you probably just want to use Safe inference. Assuming you avoid any unsafe features of the language then your modules will be marked safe. This is a benefit as now a user of your library who may want to use it as part of an API exposed to untrusted code can use the library without change. If there wasn't safety inference then either the writer of the library would have to explicitly use Safe Haskell, which is an unreasonable expectation of the whole Haskell community. Or the user of the library would have to wrap it in a shim that simply re-exported your API through a trustworthy module, an annoying practice.

7.29.6 Safe Haskell Flag Summary

In summary, Safe Haskell consists of the following three language flags:

-XSafe To be trusted, 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, every import is required to be safe regardless.

- *Module Trusted* — Yes
- *Haskell Language* — Restricted to Safe Language
- *Imported Modules* — All forced to be safe imports, all must be trusted.

-XTrustworthy 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. `-XTrustworthy` has no effect on the accepted range of Haskell programs or their semantics, except that they allow the safe import keyword.

- *Module Trusted* — Yes.
- *Module Trusted (`-fpackage-trust` enabled)* — Yes but only if the package the module resides in is also trusted.
- *Haskell Language* — Unrestricted
- *Imported Modules* — Under control of module author which ones must be trusted.

-XUnsafe Mark a module as unsafe so that it can't be imported by code compiled with `-XSafe`. 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 that the package M resides in is considered trusted for the M to be considered trusted.

And three warning flags:

-fwarn-unsafe Issue a warning if the module being compiled is regarded to be unsafe. Should be used to check the safety status of modules when using safe inference.

-fwarn-safe Issue a warning if the module being compiled is regarded to be safe. Should be used to check the safety status of modules when using safe inference.

-fwarn-trustworthy-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.

7.29.7 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-processo 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 *and* that the -fpackage-trust flag 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.

Chapter 8

Foreign function interface (FFI)

GHC (mostly) conforms to the Haskell Foreign Function Interface, whose definition is part of the Haskell Report on http://www.has

FFI support is enabled by default, but can be enabled or disabled explicitly with the `-XForeignFunctionInterface` flag.

GHC implements a number of GHC-specific extensions to the FFI Addendum. These extensions are described in Section 8.1, 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.

8.1 GHC extensions to the FFI Addendum

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.

8.1.1 Unboxed types

The following unboxed types may be used as basic foreign types (see FFI Addendum, Section 3.2): `Int#`, `Word#`, `Char#`, `Float#`, `Double#`, `Addr#`, `StablePtr#a`, `MutableByteArray#`, `ForeignObj#`, and `ByteArray#`.

8.1.2 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
```

8.1.3 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. This feature is not intended for use outside of the core libraries that come with GHC. For more details see the GHC developer wiki.

8.1.4 Interruptible foreign calls

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" :: 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`.

8.1.5 The CAPI calling convention

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 = ...
```

8.1.6 `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.

8.2 Using the FFI with GHC

The following sections also give some hints and tips on the use of the foreign function interface in GHC.

8.2.1 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 two additional files, `M_stub.c` and `M_stub.h`. GHC will automatically compile `M_stub.c` to generate `M_stub.o` at the same time.

For a plain `foreign export`, the file `M_stub.h` contains a C prototype for the foreign exported function, and `M_stub.c` contains its definition. 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);
```

and `Foo_stub.c` contains the compiler-generated definition of `foo()`. To invoke `foo()` from C, just `#include "Foo_stub.h"` and call `foo()`.

The `foo_stub.c` and `foo_stub.h` files can be redirected using the `-stubdir` option; see Section 4.7.4.

When linking the program, remember to include `M_stub.o` in the final link command line, or you'll get link errors for the missing function(s) (this isn't necessary when building your program with `ghc --make`, as GHC will automatically link in the correct bits).

8.2.1.1 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()` [1].

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`, otherwise GHC will try to link to the `Main` Haskell module.

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 Section 4.12.6), and the `-rtsopts` link-time flag overrides this. However, `-rtsopts` has no effect when `-no-hs-main` is in use (and the same goes for `-with-rtsopts`). 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"
```

[1] 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 Section 13.1.1.8.

```
#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.

8.2.1.2 Making a Haskell library that can be called from foreign code

The scenario here is much like in Section 8.2.1.1, 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.

8.2.2 Using header files

C functions are normally declared using prototypes in a C header file. Earlier versions of GHC (6.8.3 and earlier) `#include`d the header file in the C source file generated from the Haskell code, and the C compiler could therefore check that the C function being called via the FFI was being called at the right type.

GHC no longer includes external header files when compiling via C, so this checking is not performed. The change was made for compatibility with the native code generator (`-fasm`) 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.

8.2.3 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` and friends 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.

8.2.4 Multi-threading and the FFI

In order to use the FFI in a multi-threaded setting, you must use the `-threaded` option (see Section 4.12.6).

8.2.4.1 Foreign imports and multi-threading

When you call a `foreign import`ed function that is annotated as `safe` (the default), and the program was linked using `-threaded`, then the call will run concurrently with other running Haskell threads. If the program was linked without `-threaded`, 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`. 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`, make sure that the foreign code you are calling is thread-safe. In particularly, some GUI libraries are not thread-safe and require that the caller only invokes GUI methods from a single thread. If this is the case, you may need to restrict your GUI operations to a single Haskell thread, and possibly also use a bound thread (see Section 8.2.4.2).

Note that foreign calls made by different Haskell threads may execute in *parallel*, even when the `+RTS -N` flag is not being used (Section 4.15.2). The `+RTS -N` 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 Section 8.1.4 for more details.

8.2.4.2 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.

8.2.4.3 Foreign exports and multi-threading

When the program is linked with `-threaded`, then you may invoke `foreign export`ed 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 export`ed functions.

8.2.4.4 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, `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. (Unofficially, if you want to use this fast and loose version of `hs_exit()`, then call `shutdownHaskellAndExit()` instead).

8.2.5 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.

Chapter 9

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.

9.1 Source annotations

Annotations are small pragmas that allow you to attach data to identifiers in source code, which are persisted when compiled. These pieces of data can then inspected and utilized when using GHC as a library or writing a compiler plugin.

9.1.1 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 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 = ...
```

9.1.2 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 = ...
```

9.1.3 Annotating modules

You can annotate modules with the `ANN` pragma by using the `module` keyword. For example:

```
{-# ANN module (Just "A `Maybe String' annotation") #-}
```

9.2 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.

9.3 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.

9.3.1 Using compiler plugins

Plugins can be specified on the command line with the option `-fplugin=`*`module`* where *`module`* is a module in a registered package that exports a plugin. Arguments can be given to plugins with the command line option `-fplugin-opt=`*`module`*`:`*`args`*, where *`args`* are arguments interpreted by the plugin provided by *`module`*.

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 ...
$
```

Since plugins are exported by registered packages, it's safe to put dependencies on them in cabal for example, and specify plugin arguments to GHC through the `ghc-options` field.

9.3.2 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
  reinitializeGlobals
  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'.

Note carefully the `reinitializeGlobals` call at the beginning of the installation function. Due to bugs in the windows linker dealing with `libghc`, this call is necessary to properly ensure compiler plugins have the same global state as GHC at the time of invocation. Without `reinitializeGlobals`, compiler plugins can crash at runtime because they may require state that hasn't otherwise been initialized.

In the future, when the linking bugs are fixed, `reinitializeGlobals` will be deprecated with a warning, and changed to do nothing.

9.3.3 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 `CoreToDo`s and returns a list of `CoreToDo`. Before GHC begins compiling modules, it enumerates all the needed plugins you tell it to load, and runs all of their installation functions, initially on a list of passes that GHC specifies itself. After doing this for every plugin, the final list of passes is given to the optimizer, and are run by simply going over the list in order.

You should be careful with your installation function, because the list of passes you give back isn't questioned or double checked by GHC at the time of this writing. An installation function like the following:

```
install :: [CommandLineOption] -> [CoreToDo] -> CoreM [CoreToDo]
install _ _ = return []
```

is certainly valid, but also certainly not what anyone really wants.

9.3.3.1 Manipulating bindings

In the last section we saw that besides a name, a `CoreDoPluginPass` takes a pass of type `PluginPass`. A `PluginPass` is a synonym for `(ModGuts -> CoreM ModGuts)`. `ModGuts` is a type that represents the one module being compiled by GHC at any given time.

A `ModGuts` holds all of the module's top level bindings which we can examine. These bindings are of type `CoreBind` and effectively represent the binding of a name to body of code. Top-level module bindings are part of a `ModGuts` in the field `mg_binds`. Implementing a pass that manipulates the top level bindings merely needs to iterate over this field, and return a new `ModGuts` with an updated `mg_binds` field. Because this is such a common case, there is a function provided named `bindsOnlyPass` which lifts a function of type `([CoreBind] -> CoreM [CoreBind])` to type `(ModGuts -> CoreM ModGuts)`.

Continuing with our example from the last section, we can write a simple plugin that just prints out the name of all the non-recursive bindings in a module it compiles:

```
module SayNames.Plugin (plugin) where
import GhcPlugins

plugin :: Plugin
plugin = defaultPlugin {
  installCoreToDos = install
```

```
  }

install :: [CommandLineOption] -> [CoreToDo] -> CoreM [CoreToDo]
install _ todo = do
  reinitializeGlobals
  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
```

9.3.3.2 Using Annotations

Previously we discussed annotation pragmas (Section 9.1), 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, Typeable)

plugin :: Plugin
plugin = defaultPlugin {
  installCoreToDos = install
  }

install :: [CommandLineOption] -> [CoreToDo] -> CoreM [CoreToDo]
install _ todo = do
  reinitializeGlobals
  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.

9.3.4 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 Section 7.10.2).

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.

9.3.4.1 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.

Chapter 10

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 Section 1.3 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 :-!

(Section 13.1, which describes Glasgow Haskell's shortcomings vs. the Haskell language definition, may also be of interest.)

10.1 When the compiler "does the wrong thing"

"Help! The compiler crashed (or `panic'd)!" These events are *always* bugs in the GHC system—please report them.

"This is a terrible error message." If you think that GHC could have produced a better error message, please report it as a bug.

"What about this warning from the C compiler?" For example: "…warning: `Foo' declared `static' but never defined." Unsightly, but shouldn't be a problem.

Sensitivity to `.hi` interface files: GHC is very sensitive about interface files. For example, if it picks up a non-standard `Prelude.hi` file, pretty terrible things will happen. If you turn on `-XNoImplicitPrelude`, 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`; this causes a "lint" pass to check for errors (notably type errors) after each Core-to-Core transformation pass. We run with `-dcore-lint` on all the time; it costs about 5% in compile time.

"Why did I get a link error?" If the linker complains about not finding `_<something>_fast`, then something is inconsistent: you probably didn't compile modules in the proper dependency order.

"Is this line number right?" On this score, GHC usually does pretty well, especially if you "allow" it to be off by one or two. In the case of an instance or class declaration, the line number may only point you to the declaration, not to a specific method.

Please report line-number errors that you find particularly unhelpful.

10.2 When your program "does the wrong thing"

(For advice about overly slow or memory-hungry Haskell programs, please see Chapter 6).

"Help! My program crashed!" (e.g., a `segmentation fault' or `core dumped')

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`. 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 Section 4.7.12.

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 Section 1.3).

"What's with this `arithmetic (or `floating') exception' "? `Int`, `Float`, and `Double` arithmetic is *unchecked*. Overflows, underflows and loss of precision are either silent or reported as an exception by the operating system (depending on the platform). Divide-by-zero *may* cause an untrapped exception (please report it if it does).

Chapter 11

Other Haskell utility programs

This section describes other program(s) which we distribute, that help with the Great Haskell Programming Task.

11.1 "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.

11.2 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.

11.2.1 command line syntax

hsc2hs takes input files as arguments, and flags that modify its behavior:

`-o FILE` or `--output=FILE` Name of the Haskell file.

`-t FILE` or `--template=FILE` The template file (see below).

`-c PROG` or `--cc=PROG` The C compiler to use (default: **gcc**)

`-l PROG` or `--ld=PROG` The linker to use (default: **gcc**).

`-C FLAG` or `--cflag=FLAG` An extra flag to pass to the C compiler.

-I DIR Passed to the C compiler.

-L FLAG or --lflag=FLAG An extra flag to pass to the linker.

-i FILE or --include=FILE As if the appropriate #include directive was placed in the source.

-D NAME[=VALUE] or --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 or --keep-files Proceed as normal, but do not delete any intermediate files.

-x or --cross-compile Activate cross-compilation mode (see Section 11.2.4).

--cross-safe Restrict the .hsc directives to those supported by the --cross-compile mode (see Section 11.2.4). 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.

-? or --help Display a summary of the available flags and exit successfully.

-V or --version Output version information and exit successfully.

The input file should end with .hsc (it should be plain Haskell source only; literate Haskell is not supported at the moment). Output files by default get names with the .hsc suffix replaced:

.hs	Haskell file
_hsc.h	C header
_hsc.c	C file

The C program is compiled using the Haskell compiler. This provides the include path to HsFFI.h which is automatically included into the C program.

11.2.2 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, #warning 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`.

#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.

11.2.3 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.

11.2.4 Cross-compilation

hsc2hs normally operates by creating, compiling, and running a C program. That approach doesn't work when cross-compiling -- in this case, the C compiler's generates code for the target machine, not the host machine. For this situation, there's a special mode **hsc2hs --cross-compile** which can generate the .hs by extracting information from compilations only -- specifically, whether or not compilation fails.

Only a subset of `.hsc` syntax is supported by `--cross-compile`. The following are unsupported:

- `#{const_str}`
- `#{let}`

- `#{def}`
- Custom constructs

Chapter 12

Running GHC on Win32 systems

12.1 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".

12.2 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.

12.3 Interacting with the terminal

By default GHC builds applications that open a console window when they start. If you want to build a GUI-only application, with no console window, use the flag `-optl-mwindows` in the link step.

Warning: Windows GUI-only programs have no stdin, stdout or stderr so using the ordinary Haskell input/output functions will cause your program to fail with an IO exception, such as:

```
    Fail: <stdout>: hPutChar: failed (Bad file descriptor)
```

However using Debug.Trace.trace is alright because it uses Windows debugging output support rather than stderr.

For some reason, Mingw ships with the `readline` library, but not with the `readline` headers. As a result, GHC (like Hugs) does not use `readline` for interactive input on Windows. You can get a close simulation by using an emacs shell buffer!

12.4 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.

12.5 Using GHC (and other GHC-compiled executables) with cygwin

12.5.1 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`).

12.5.2 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.

12.5.3 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 ``ROOT=`pwd` `` in makefile hierarchies or configure scripts), cygwin provides a tool called **cygpath** that can convert cygwin's unix-style paths to their actual windows-style counterparts. Many cygwin tools actually accept absolute windows-style paths (remember, though, that you either need to escape '\' or convert '\' to '/'), so you should be fine just using those everywhere. If you need to use tools that do some kind of path-mangling that depends on unix-style paths (one fun example is trying to interpret ':' as a separator in path lists..), you can still try to convert paths using **cygpath** just before they are passed to GHC and friends.
- If you don't have **cygpath**, you probably don't have cygwin and hence no problems with it... unless you want to write one build process for several platforms. Again, relative paths are your friend, but if you have to use absolute paths, and don't want to use different tools on different platforms, you can simply write a short Haskell program to print the current directory (thanks to George Russell for this idea): compiled with GHC, this will give you the view of the file system that GHC depends on (which will differ depending on whether GHC is compiled with cygwin's gcc or mingw's gcc or on a real unix system..) - that little

program can also deal with escaping '\' in paths. Apart from the banner and the startup time, something like this would also do:

```
$ echo "Directory.getCurrentDirectory >>= putStrLn . init . tail . show " | ghci
```

12.6 Building and using Win32 DLLs

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 Section 4.13.

- 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.

12.6.1 Creating a DLL

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.

12.6.2 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 Section 8.2.1.2; 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 bug #3605).

12.6.2.1 Using from VBA

An example of using `Adder.dll` from VBA is:

```
Private Declare Function Adder Lib "Adder.dll" Alias "adder@8" _
      (ByVal x As Long, ByVal y As Long) As Long

Private Declare Sub HsStart Lib "Adder.dll" ()
Private Declare Sub HsEnd Lib "Adder.dll" ()

Private Sub Document_Close()
```

```
HsEnd
End Sub

Private Sub Document_Open()
HsStart
End Sub

Public Sub Test()
MsgBox "12 + 5 = " & Adder(12, 5)
End Sub
```

This example uses the `Document_Open`/`Close` functions of Microsoft Word, but provided `HsStart` is called before the first function, and `HsEnd` after the last, then it will work fine.

12.6.2.2 Using from C++

An example of using `Adder.dll` from C++ is:

```
// Tester.cpp
#include "HsFFI.h"
#include "Adder_stub.h"
#include <stdio.h>

extern "C" {
    void HsStart();
    void HsEnd();
}

int main()
{
    HsStart();
    // can now safely call functions from the DLL
    printf("12 + 5 = %i\n", adder(12,5))    ;
    HsEnd();
    return 0;
}
```

This can be compiled and run with:

```
$ ghc -o tester Tester.cpp Adder.dll.a
$ tester
12 + 5 = 17
```

Chapter 13

Known bugs and infelicities

13.1 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 (Chapter 10) for information about crashes, space leaks, and other undesirable phenomena.

The limitations here are listed in Haskell Report order (roughly).

13.1.1 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.

13.1.1.1 Lexical syntax

- Certain lexical rules regarding qualified identifiers are slightly different in GHC compared to the Haskell report. When you have *module.reservedop*, such as `M.\`, GHC will interpret it as a single qualified operator rather than the two lexemes `M` and `.\`.

13.1.1.2 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)
```

13.1.1.3 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`.

13.1.1.4 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 Section 7.4.2.

13.1.1.5 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 Section 4.7.9. 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."

13.1.1.6 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 superclasses The `Bits` class does not have a `Num` superclasses. It therefore does not have default methods for the `bit`, `testBit` and `popCount` methods.

You can make code that works with both Haskell2010 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.

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.

13.1.1.7 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

`zip` and `zipWith` semantics

`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`.

13.1.1.8 The Foreign Function Interface

`hs_init()` not allowed after `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.

13.1.2 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.

The `Char` type Following the ISO-10646 standard, `maxBound :: Char` in GHC is `0x10FFFF`.

Sized integral types 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` function (and hence also `fromIntegral`) is a special case when converting to `Int`. The value of `fromIntegral x :: Int` is given by taking the lower n bits of `(abs x)`, multiplied by the sign of `x` (in 2's complement n-bit arithmetic). This behaviour was chosen so that for example writing `0xffffffff :: Int` preserves the bit-pattern in the resulting `Int`.

Negative literals, such as `-3`, are specified by (a careful reading of) the Haskell Report as meaning `Prelude.negate (Prelude.fromInteger 3)`. So `-2147483648` means `negate (fromInteger 2147483648)`. Since `fromInteger` takes the lower 32 bits of the representation, `fromInteger (2147483648::Integer)`, computed at type `Int` is `-2147483648::Int`. The `negate` operation then overflows, but it is unchecked, so `negate (-2147483648::Int)` is just `-2147483648`. In short, one can write `minBound::Int` as a literal with the expected meaning (but that is not in general guaranteed).

The `fromIntegral` function also preserves bit-patterns when converting between the sized integral types (`Int8`, `Int16`, `Int32`, `Int64` and the unsigned `Word` variants), see the modules `Data.Int` and `Data.Word` in the library documentation.

Unchecked float 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).

13.2 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.

13.2.1 Bugs in GHC

- GHC can warn about non-exhaustive or overlapping patterns (see Section 4.8), and usually does so correctly. But not always. It gets confused by string patterns, and by guards, and can then emit bogus warnings. The entire overlap-check code needs an overhaul really.
- 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)
```

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` 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` option (see Section 4.16, 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.

13.2.2 Bugs in GHCi (the interactive GHC)

- GHCi does not respect the `default` declaration in the module whose scope you are in. Instead, for expressions typed at the command line, you always get the default default-type behaviour; that is, `default(Int,Double)`.

  It would be better for GHCi to record what the default settings in each module are, and use those of the 'current' module (whatever that is).

- On Windows, there's a GNU ld/BFD bug whereby it emits bogus PE object files that have more than 0xffff relocations. When GHCi tries to load a package affected by this bug, you get an error message of the form

```
Loading package javavm ... linking ... WARNING: Overflown relocation field (# relocs found ↩
    : 30765)
```

  The last time we looked, this bug still wasn't fixed in the BFD codebase, and there wasn't any noticeable interest in fixing it when we reported the bug back in 2001 or so.

  The workaround is to split up the .o files that make up your package into two or more .o's, along the lines of how the "base" package does it.

Chapter 14

Index